คำนิยม *หนังสือคู่มือ RISC-V*

หนังสือที่ออกมาทันเวลาเล่มนี้ อธิบายอย่างกระชับถึง ISA ของ RISC-V ที่เรียบง่าย ฟรี และเปิดกว้าง ซึ่งกำลังได้รับการยอมรับอย่างรวดเร็วในหลายภาคส่วนของการคำนวณ หนังสือเล่มนี้ยังมีข้อมูลเชิงลึกมากมายเกี่ยวกับสถาปัตยกรรมคอมพิวเตอร์โดยทั่วไป รวมถึงอธิบายการตัดสินใจออกแบบเฉพาะที่เราเลือกในการสร้าง RISC-V ฉันสามารถจินตนาการได้ว่าหนังสือเล่มนี้จะกลายเป็นคู่มืออ้างอิงที่ถูกใช้งานบ่อยสำหรับผู้ปฏิบัติงาน RISC-V หลายคน

—ศาสตราจารย์ Krste Asanovi'c, มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ หนึ่งในสี่สถาปนิกของ RISC-V

ฉันชอบ RISC-V และหนังสือเล่มนี้ เพราะมันสง่างาม—สั้น กระชับ และครบถ้วน คำอธิบายประกอบของหนังสือให้ประวัติศาสตร์ แรงจูงใจ และการวิจารณ์สถาปัตยกรรมโดยไม่จำเป็น

—C. Gordon Bell, Microsoft และผู้ออกแบบสถาปัตยกรรมชุดคำสั่ง Digital PDP-11 และ VAX-11

หนังสือเล่มเล็กที่สะดวกนี้สรุปองค์ประกอบสำคัญทั้งหมดของสถาปัตยกรรมชุดคำสั่ง RISC-V ได้อย่างง่ายดาย เป็นคู่มืออ้างอิงที่สมบูรณ์แบบสำหรับทั้งนักศึกษาและผู้ปฏิบัติงาน

—ศาสตราจารย์ Randy Katz, มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ หนึ่งในผู้ประดิษฐ์ระบบจัดเก็บข้อมูล RAID

RISC-V เป็นตัวเลือกที่ดี สำหรับนักเรียนในการเรียนรู้สถาปัตยกรรมชุดคำสั่ง และการเขียนโปรแกรมระดับแอสเซมบลี ซึ่งเป็นพื้นฐานสำหรับงานในภาษาระดับสูงในภายหลัง หนังสือที่เขียนอย่างชัดเจนเล่มนี้ เป็นเป็นบทนำที่ดีเกี่ยวกับ RISC-V เสริมด้วยความคิดเห็นเชิงลึกเกี่ยวกับประวัติวิวัฒนาการ และการเปรียบเทียบกับสถาปัตยกรรมทั่วไปอื่นๆ โดยอิงจากจากประสบการณ์กับสถาปัตยกรรมอื่นๆ ในอดีต ผู้ออกแบบ RISC-V สามารถหลีกเลี่ยงคุณสมบัติที่ไม่จำเป็น และไม่สม่ำเสมอได้ ซึ่งทำให้คำสั่งง่ายขึ้น แม้ว่าจะเรียบง่าย แต่ก็ทรงพลังเพียงพอสำหรับการใช้งานจริงในแอปพลิเคชันที่หลากหลาย เมื่อนานมาแล้ว ฉันเคยสอนคอร์สแรกเกี่ยวกับการเขียนโปรแกรมแอสเซมบลี และถ้าฉันทำแบบนั้นตอนนี้ ฉันจะใช้หนังสือเล่มนี้อย่างมีความสุข

—John Mashey หนึ่งในนักออกแบบสถาปัตยกรรมชุดคำสั่ง MIPS

หนังสือเล่มนี้บอกว่า RISC-V สามารถทำอะไรได้บ้าง และทำไมผู้ออกแบบถึงเลือกที่จะมอบความสามารถเหล่านั้นให้กับมัน ที่น่าสนใจยิ่งกว่านั้น ผู้เขียนบอกว่าเหตุใด RISC-V ยกเลิกบางคุณสมบัติ ที่มีอยู่ในเครื่องยุคแรกๆ เหตุผลเหล่านี้น่าสนใจพอๆ กับสิ่งที่ RISC-V จึงละทิ้งสิ่งที่พบในเครื่องรุ่นก่อนหน้า เหตุผลเหล่านี้น่าสนใจไม่แพ้ความสามารถและสิ่งที่ RISC-V ละทิ้ง

—Ivan Sutherland ผู้ได้รับรางวัล Turing Award ได้รับการยกย่องให้เป็นบิดาแห่งคอมพิวเตอร์กราฟิก

RISC-V จะเปลี่ยนโลก และหนังสือเล่มนี้จะช่วยให้คุณเป็นส่วนหนึ่งของการเปลี่ยนแปลงนั้น

—ศาสตราจารย์ Michael B. Taylor มหาวิทยาลัยวอชิงตัน

หนังสือเล่มนี้ จะเป็นข้อมูลอ้างอิงที่มีค่า สำหรับทุกคนที่ทำงานกับ ISA ของ RISC-V รหัสดำเนินการนำเสนอในรูปแบบที่มีประโยชน์หลายรูปแบบ สำหรับการอ้างอิงอย่างรวดเร็ว ทำให้การเขียนโค้ดแอสเซมบลีและการตีความทำได้ง่ายขึ้น นอกจากนี้ คำอธิบายและตัวอย่างวิธีการใช้ ISA ทำให้งานของโปรแกรมเมอร์ทำงานง่ายยิ่งขึ้น การเปรียบเทียบกับ ISA อื่นๆ นั้นน่าสนใจ และแสดงให้เห็นว่าทำไมผู้สร้าง RISC-V ตัดสินใจออกแบบเช่นนั้น

—Megan Wachs, ดุษฎีบัณฑิต, วิศวกร SiFive

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	B	BEQ rs1,rs2,imm	
	Branch ≠	B	BNE rs1,rs2,imm	
	Branch <	B	BLT rs1,rs2,imm	
	Branch ≥	B	BGE rs1,rs2,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm	
Jump & Link	J&L	J	JAL rd,imm	
	Jump & Link Register	I	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

Control Status Register (CSR)			
Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads				
Load Byte	I	LB	rd,rs1,imm	
Load Halfword	I	LH	rd,rs1,imm	
Load Byte Unsigned	I	LBU	rd,rs1,imm	
Load Half Unsigned	I	LHU	rd,rs1,imm	
Load Word	I	LW	rd,rs1,imm	
Stores Store Byte	S	SB	rs1,rs2,imm	
Store Halfword	S	SH	rs1,rs2,imm	
Store Word	S	SW	rs1,rs2,imm	SD rs1,rs2,imm

+RV64I

LWU	rd,rs1,imm	
LD	rd,rs1,imm	

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Float Load Word	CL	C.FLW rd',rs1',imm	LW rd',rs1',imm*8
	Float Load Word SP	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8
	Float Load Double	CL	C.FLD rd',rs1',imm	FLD rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Float Store Word	CS	C.FSW rs1',rs2',imm	FSW rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8
	Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
	AND	CR	C.AND rd,rs1	AND rd,rd,rs1
	AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm
	OR	CR	C.OR rd,rs1	OR rd,rd,rs1
	eXclusive OR	CR	C.XOR rd,rs1	AND rd,rd,rs1
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

Optional Compressed Extention: RV64C

All RV32C (except C.JAL*, 4 word loads, 4 word stores) plus:*

- ADD Word (C.ADDW)
- ADD Imm. Word (C.ADDIW)
- SUBtract Word (C.SUBW)
- Load Doubleword (C.LD)
- Load Doubleword SP (C.LDSP)
- Store Doubleword (C.SD)
- Store Doubleword SP (C.SDSP)

32-bit Instruction Formats

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

16-bit (RVC) Instruction Formats

	15–14–13–12	11–10	9–8–7	6–5–4–3–2	1–0
CR	funct4	rd/rs1		rs2	op
CI	funct3 | imm	rd/rs1		imm	op
CSS	funct3	imm		rs2	op
CIW	funct3	imm		rd′	op
CL	funct3	imm	rs1′	imm | rd′	op
CS	funct3	imm	rs1′	imm | rs2′	op
CB	funct3	offset	rs1′	offset	op
CJ	funct3	jump target			op

RISC-V Integer Base (RV32I/64I), privileged, and optional compressed instruction extension (RV32/64C). Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 bits wide in RV64I (x0=0). RV64I adds 12 instructions for 64-bit data.

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

3 Optional Floating-Point Instruction Extensions: Zfh, RVF, & RVD

Category	Name	Fmt	RV32{H|F|D} (HP,SP,DP)		+RV64{H|F|D}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{H|S|D}.W	rd,rs1	FCVT.{H|S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{H|S|D}.WU	rd,rs1	FCVT.{H|S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{H|S|D}	rd,rs1	FCVT.L.{H|S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{H|S|D}	rd,rs1	FCVT.LU.{H|S|D}	rd,rs1
Load	Load	I	FL{H,W,D}	rd,rs1,imm		
Store	Store	S	FS{H,W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{H|S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{H|S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{H|S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{H|S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{H|S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{H|S|D}	rd,rs1/2/3		
	Multiply-SUBtract	R	FMSUB.{H|S|D}	rd,rs1/2/3		
	Negative Multiply-SUBtract	R	FNMSUB.{H|S|D}	rd,rs1/2/3		
	Negative Multiply-ADD	R	FNMADD.{H|S|D}	rd,rs1/2/3		
Sign Inject	SiGN source	R	FSGNJ.{H|S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{H|S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{H|S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{H|S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{H|S|D}	rd,rs1,rs2		
Compare	compare Float=	R	FEQ.{H|S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{H|S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{H|S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{H|S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-a7	Caller
x18-27	s2-s11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-6,ft0-11	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

Optional Vector Extension: RVV

RV32/64V in Figure 8.1. All names really start with "vector". Using regex notation, {_|F} means integer and FP versions: e.g., V{_|F}ADD is VADD and VFADD. Similarly, VREM{_|U} represents VREM and VREMU, the signed and Unsigned versions. For data transfer instructions, {8-64} means 4 widths: e.g., VL{8-64} is VL8, VL16, VL32, and VL64.

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	VSETVL	rd,rs1
SET VL Imm.	R	VSETVLI	rd,rs1
SET Imm. VL	R	VSETIVL	rd,rs1
Load	I	VL{8-64}	rd,rs1,imm
Load Strided	R	VLS {8-64}	rd,rs1,rs2
Load Ord. indeX	R	VLOX{8-64}	rd,rs1,rs2
Load Uno. indeX	R	VLUX{8-64}	rd,rs1,rs2
Store	S	VS{8-64}	rd,rs1,imm
Store Strided	R	VSS {8-64}	rd,rs1,rs2
Store Ord. indeX	R	VSOX{8-64}	rd,rs1,rs2
Store Uno. indeX	R	VSUX{8-64}	rd,rs1,rs2
Mask =	R	VM{S|F}EQ	rd,rs1,rs2
Mask ≠	R	VM{S|F}NE	rd,rs1,rs2
Mask <	R	VM{S|F}LT{_|U}	rd,rs1,rs2
Mask ≤	R	VM{S|F}LE{_|U}	rd,rs1,rs2
Mask >	R	VM{S|F}GT{_|U}	rd,rs1,rs2
Mask ≥	R	VM{S|F}GE{ |U}	rd,rs1,rs2
Shift Left Log.	R	SLL	rd,sr1,rs2
Shift Right Log.	R	SRL	rd,sr1,rs2
Shift Right Arith.	R	SRA	rd,sr1,rs2
XOR	R	VXOR	rd,sr1,rs2
AND	R	VAND	rd,sr1,rs2
OR	R	VOR	rd,sr1,rs2
ADD	R	V{_|F}ADD	rd,rs1,rs2
SUBtract	R	V{_|F}SUB	rd,rs1,rs2
MULtiply	R	V{_|F}MUL	rd,rs1,rs2
DIVide	R	V{_|F}DIV{_|U}	rd,rs1,rs2
SQuare RooT	R	V{_|F}SQRT	rd,rs1,rs2
MULtiply-ADD	R	V{_|F}MADD	rd,rs1,rs2
MULtiply-SUB	R	V{_|F}MSUB	rd,rs1,rs2
Neg. MUL-ADD	R	V{_|F}NMSUB	rd,rs1,rs2
Neg. MUL-SUB	R	V{ |F}NMADD	rd,rs1,rs2
Fl. Pt. Sign inJect	R	VFSJ	rd,sr1,rs2
FSJ Negative	R	VFSJN	rd,sr1,rs2
FSJ eXclusive or	R	VFSJX	rd,sr1,rs2
MINimum	R	V{_|F}MIN{_|U}	rd,rs1,rs2
MAXimum	R	V{ |F}MAX{ |U}	rd,rs1,rs2
MULtiply High	R	VMULH{_|U}	rd,rs1,rs2
REMainder	R	VREM{_|U}	rd,rs1,rs2
Reverse SUB	R	V{_|F}RSUB	rd,rs1,rs2
Reverse DIV	R	V{_|F}RDIV	rd,rs1,rs2
Mask XOR	R	VMXOR	rd,sr1,rs2
Mask AND	R	VMAND	rd,sr1,rs2
Mask OR	R	VMOR	rd,sr1,rs2
Mask XNOR	R	VMXNOR	rd,sr1,rs2
Mask NAND	R	VMNAND	rd,sr1,rs2
Mask NOR	R	VMNOR	rd,sr1,rs2
Mask AND NOT	R	VMANDNOT	rd,sr1,rs2
Mask OR NOT	R	VMORNOT	rd,sr1,rs2

Calling convention and 6 optional extensions: 8 RV32M (Multiply-divide) instructions; 11 RV32A (Atomic); 49 RV32V (Vector); and 34 floating-point instructions each for 16-, 32-, and 64-bit data (Zfh, RV32F, RV32D). For multiply-add operations, "rs1/2/3" represents "rs1, rs2, rs3". Using regex notation, FADD.{F|D} is both FADD.F and FADD.D. RV32F and RV32 add floating-point registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. RV32V adds vector registers v0-v31, vector predicate registers vp0-vp7, and a vector length register vl. RV64 adds a few more instructions: RVM adds 4, RVA 11, Zfh 6, RVF 6, RVD 6, and RVV 0.

คู่มือ RISC-V:
สถาปัตยกรรมแบบเปิด

ฉบับปรับปรุงที่ 2, 2.0.0

เดวิด แพตเตอร์สัน และ แอนดรูว์ วอเตอร์แมน

แปลโดย พอล เชอร์แมน และ นฤมล รัตนคุณากร

26 สิงหาคม 2568

ลิขสิทธิ์ พ.ศ. ๒๕๖๕ Strawberry Canyon จำกัด สงวนลิขสิทธิ์

ห้ามทำซ้ำส่วนใดส่วนหนึ่งของหนังสือเล่มนี้หรือวัสดุที่เกี่ยวข้องในรูปแบบใดๆ โดยไม่ได้รับความยินยอมเป็นลายลักษณ์อักษรจากผู้ถือลิขสิทธิ์

พิมพ์ครั้งที่: 2.0.0

พื้นหลังของหน้าปกเป็นภาพของ ***โมนา ลิซ่า*** เป็นภาพเหมือนของ Lisa Gherardini วาดโดย Leonardo da Vinci ระหว่างปี พ.ศ. 2040 ถึง 2049 ซึ่งกษัตริย์แห่งฝรั่งเศสได้ทรงซื้อภาพนี้จาก เลโอนาร์โด เมื่อประมาณ ปี พ.ศ. 2073 และจัดแสดงที่พิพิธภัณฑ์ลูฟร์ในปารีส ตั้งแต่ปี พ.ศ. 2340 โมนาลิซ่าถือเป็นงาน ศิลปะที่รู้จักกันดีที่สุดในโลก โมนาลิซ่าเป็นตัวแทนของความสง่างาม ซึ่งเราเชื่อว่าเป็นคุณสมบัติของ RISC-V

หนังสือเล่มนี้จัดทำด้วย LaTeX Makefiles ไฟล์สไตล์ และสคริปต์ส่วนใหญ่ที่จำเป็นมีให้ใช้งานภายใต้ใบอนุญาต BSD ที่ `github.com/armandofox/latex2ebook`

Arthur Klepchukov ออกแบบหน้าปกและกราฟิกสำหรับทุกเวอร์ชัน

ข้อมูลรายการบรรณานุกรม

Names: Patterson, David A. | Waterman, Andrew, 1986-
Title: The RISC-V reader: an open architecture atlas
David Patterson and Andrew Waterman.
Description: First edition. | [Berkeley, California] : Strawberry Canyon LLC, 2017. |
Includes bibliographical references and index.
Identifiers: ISBN 978-0-9992491-1-6
Subjects: LCSH: Computer architecture. | RISC microprocessors. |
Assembly languages (Electronic computers)
Classification: LCC QA76.9.A73 P38 2017 | DDC 004.22- -dc23

คำอุทิศ



เดวิด แพตเตอร์สัน อุทิศหนังสือเล่มนี้ให้กับพ่อ-แม่ของเขา

—ถึงพ่อของฉัน เดวิด ผู้ซึ่งฉันได้รับความคิดสร้างสรรค์ ความสามารถด้านกีฬา และความกล้าหาญในการต่อสู้เพื่อสิ่งที่ถูกต้อง และ

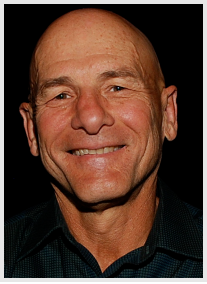
—ถึงแม่ของฉัน ลูซี่ ผู้ซึ่งฉันได้รับสติปัญญา ความมองโลกในแง่ดี และอุปนิสัยใจคอของฉันมาจากท่าน

ขอบคุณที่เป็นแบบอย่างที่ดีเยี่ยม ซึ่งสอนฉันถึงความหมายของการเป็นคู่ครอง พ่อแม่ และปู่ย่าตายายที่ดี



แอนดรูว์ วอเตอร์แมน อุทิศหนังสือเล่มนี้ให้กับพ่อแม่ของเขา จอห์น และเอลิซาเบธ ผู้ซึ่งให้การสนับสนุนอย่างมาก แม้ว่าจะอยู่ห่างออกไปหลายพันไมล์

เกี่ยวกับผู้เขียน

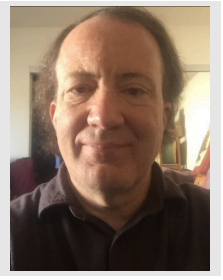


เดวิด แพตเตอร์สัน เกษียณในปี 2016 หลังจากทำงานเป็นศาสตราจารย์ด้านวิทยาการคอมพิวเตอร์ที่ UC Berkeley นาน 40 ปี จากนั้นเขาเข้าร่วมกับ Google Brain ในตำแหน่งวิศวกรอาวุโส เขายังดำรงตำแหน่งรองประธานคณะกรรมการของ RISC-V International และผู้อำนวยการห้องปฏิบัติการโอเพนซอร์ส RISC-V International ในอดีต เขาเคยเป็นประธานของภาควิชาวิทยาการคอมพิวเตอร์ที่ Berkeley และได้รับเลือกให้เป็นประธานของ Computing Research Association (CRA) และประธานของ Association for Computing Machinery (ACM) ในช่วงทศวรรษ 1980 เขาเป็นผู้นำโครงการ Reduced Instruction Set Computer (RISC) สี่รุ่น ซึ่งเป็นแรงบันดาลใจให้ตั้งชื่อ RISC รุ่นล่าสุดของ Berkeley ว่า “RISC Five” ร่วมกับแอนดรูว์ วอเตอร์แมน ซึ่งเขาเป็นหนึ่งในสี่สถาปนิกของ RISC-V นอกเหนือจาก RISC แล้ว โครงการวิจัยที่โด่งดังที่สุดของเขาคือ Redundant Arrays of Inexpensive Disks (RAID) งานวิจัยนี้ทำให้เขาเขียนบทความจำนวนมาก หนังสือ 7 เล่ม และได้รับรางวัลกว่า 40 รางวัล รวมถึงการได้รับเลือกเป็นสมาชิกของ National Academy of Engineering, National Academy of Sciences และ Silicon Valley Engineering Hall of Fame เขาได้รับรางวัล Distinguished Service Awards จาก ACM, CRA และ SIGARCH รวมถึงรางวัลด้านการศึกษา เช่น รางวัล รางวัลการสอนยอดเยี่ยมจาก UC Berkeley, รางวัล Karlstrom Outstanding Educator Award จาก ACM, รางวัล Mulligan Education Medal จาก IEEE และรางวัล Textbook Excellence Awards 2 รางวัล จาก Text and Academic Authors Association รางวัลล่าสุดของเขาคือรางวัล Tapia Achievement Award for Scientific Scholarship, Civic Science, and Diversifying Computing, รางวัล BBVA Foundation Frontiers of Knowledge Award และรางวัล ACM A.M. Turing Award ซึ่งสองรางวัลหลังนี้เขาได้รับร่วมกับจอห์น เฮนเนสซี่ เขาสำเร็จการศึกษาทุกปริญญาจาก UCLA ซึ่งมอบรางวัล Outstanding Engineering Academic Alumni Award ให้เขา เขาเติบโตในแคลิฟอร์เนียตอนใต้ และในเวลาว่าง เขาชอบเล่นบอดี้เซิร์ฟ ปั่นจักรยาน เล่นฟุตบอลกับลูกชาย และเดินป่ากับภรรยา ซึ่งเป็นรักแรกสมัยมัธยม และในปี 2022 พวกเขาฉลองครบรอบแต่งงาน 55 ปี

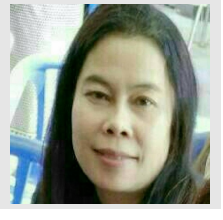


แอนดรูว์ วอเตอร์แมน ดำรงตำแหน่งหัวหน้าวิศวกรและผู้ร่วมก่อตั้ง SiFive โดย SiFive ก่อตั้งโดยผู้สร้างสถาปัตยกรรม RISC-V เพื่อจัดหาชิปแบบกำหนดเองที่มีต้นทุนต่ำ โดยใช้ RISC-V แอนดรูว์ได้รับปริญญาเอกด้านวิทยาการคอมพิวเตอร์จาก UC Berkeley ซึ่งในระหว่างนั้น เขารู้สึกเบื่อหน่ายกับความไม่แน่นอนของสถาปัตยกรรมชุดคำสั่งที่มีอยู่ เขาจึงได้ร่วมออกแบบสถาปัตยกรรมชุดคำสั่ง RISC-V และไมโครโปรเซสเซอร์ RISC-V รุ่นแรก เขาเป็นหนึ่งในผู้มีส่วนสำคัญในการสร้างชิป Rocket ที่ใช้ RISC-V แบบโอเพ่นซอร์ส, ภาษาการสร้างฮาร์ดแวร์ Chisel, และการพอร์ต RISC-V สำหรับเคอร์เนลระบบปฏิบัติการ Linux และคอมไพเลอร์ GNU C และไลบรารี C นอกจากนี้ เขายังได้รับปริญญาโทจาก UC Berkeley ซึ่งเป็นพื้นฐานของส่วนขยาย RVC สำหรับ RISC-V และปริญญาตรีจาก Duke University

เกี่ยวกับผู้แปล



พอล เซอร์แมน เป็นวิศวกรคอมพิวเตอร์ใน Silicon Valley ที่มีความห่วงใยต่อปัญหาของผู้คนทั่วโลก เขามีบทบาทสำคัญในการพัฒนาอุตสาหกรรมการจัดเก็บข้อมูลในช่วง 25 ปีที่ผ่านมา โดยทำงานกับบริษัทอย่าง Western Digital และ Seagate เขาได้ตีพิมพ์บทความจำนวนมากกับชุมชนผู้ใช้ SAS Institute เกี่ยวกับปัญหาทางสถิติ เขาร่วมเขียนบทความและหนังสือหลายเล่ม เกี่ยวกับเศรษฐศาสตร์แบบก้าวหน้า และได้รับสิทธิบัตรสหรัฐอเมริกาสี่ฉบับ สำหรับการประดิษฐ์เกี่ยวกับการทดสอบและการผลิตอุปกรณ์จัดเก็บข้อมูล เขาเป็นผู้เผยแพร่ MRAM และได้แสดงให้เห็นว่า RISC-V ใช้งานง่ายเหมือน PI ในงาน Flash Memory Summit เดือนสิงหาคม 2022 เขาสำเร็จการศึกษาระดับปริญญาโทสาขาฟิสิกส์จาก University of California Irvine และสำเร็จการศึกษาระดับปริญญาตรีจาก University of California Santa Cruz



นฤมล รัตนคุณากร เป็นนักวิทยาศาสตร์คอมพิวเตอร์ที่รักทุกสิ่งที่เรียบง่ายและสง่างาม เธอมีประสบการณ์หลายปีในอุตสาหกรรมนี้ เธอสำเร็จการศึกษาระดับปริญญาตรีจากมหาวิทยาลัยขอนแก่น

สารบัญย่อ

การ์ดอ้างอิง RISC-V 2
สารบัญรูปภาพ xiv
คำนำ xvi
1 ทำไมต้อง RISC-V ? 2
2 RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V 18
3 ภาษาแอสเซมบลี RISC-V 38
4 RV32M: การคูณและการหาร 52
5 RV32FD: จุดลอยตัวแบบความแม่นยำเดี่ยวและคู่ 56
6 RV32A: คำสั่งอะตอมิก 70
7 RV32C: คำสั่งแบบบีบอัด 74
8 RV32V: เวกเตอร์ 82
9 RV32B: การจัดการบิต 122
10 RV64: คำสั่งที่อยู่หน่วยความจำ 64 บิต 126
11 RV32/64 สถาปัตยกรรมสิทธิพิเศษ 142
12 ส่วนขยายเพิ่มเติม RISC-V ในอนาคต 164
A รายการคำสั่ง RISC-V 166
B การแปลงคำสั่ง RISC-V 222
C การแปลภาษาไทย 230

ดัชนี 246

สารบัญ

การ์ดอ้างอิง RISC-V **2**

สารบัญรูปภาพ **xiv**

คำนำ **xvi**

1 ทำไมต้อง RISC-V ? **2**

1.1 บทนำ 2
1.2 ISA แบบโมดูลาร์ เทียบกับ ISA แบบเพิ่มขยาย 3
1.3 การออกแบบ ISA 101 6
1.4 ภาพรวมของหนังสือเล่มนี้ 13
1.5 ข้อสรุปท้ายบท 15
1.6 เรียนรู้เพิ่มเติม 16

2 RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V **18**

2.1 บทนำ 18
2.2 รูปแบบคำสั่ง RV32I 18
2.3 รีจิสเตอร์ RV32I 22
2.4 การคำนวณจำนวนเต็มของ RV32I 24
2.5 โหลดและจัดเก็บข้อมูลใน RV32I 25
2.6 การกระโดดแบบมีเงื่อนไขใน RV32I 27
2.7 การกระโดดแบบไม่มีเงื่อนไขใน RV32I 28
2.8 ส่วนเบ็ดเตล็ดของ RV32I 29
2.9 เปรียบเทียบ RV32I, ARM-32, MIPS-32, และ x86-32 30
2.10 ข้อสังเกตสรุป 30
2.11 เรียนรู้เพิ่มเติม 31

3 ภาษาแอสเซมบลี RISC-V **38**
3.1 บทนำ 38
3.2 ข้อตกลงการเรียกฟังก์ชัน 38
3.3 แอสเซมบลี 41
3.4 ตัวเชื่อมโยง (Linker) 45
3.5 การเชื่อมโยงแบบคงที่เทียบกับแบบไดนามิก 48
3.6 โหลดเดอร์ (Loader) 49
3.7 ข้อสังเกตสรุป 49
3.8 เรียนรู้เพิ่มเติม 50

4 RV32M: การคูณและการหาร **52**
4.1 บทนำ 52
4.2 ข้อสังเกตสรุป 54
4.3 เรียนรู้เพิ่มเติม 55

5 RV32FD: จุดลอยตัวแบบความแม่นยำเดี่ยวและคู่ **56**
5.1 บทนำ 56
5.2 รีจิสเตอร์จุดลอยตัว 56
5.3 การโหลด การบันทึก และการคำนวณจุดลอยตัว 59
5.4 การย้ายและแปลงข้อมูลจุดลอยตัว 60
5.5 คำสั่งจุดลอยตัวสำหรับไลบรารีคณิตศาสตร์ 61
5.6 ส่วนขยายมาตรฐาน “Q” สำหรับจุดลอยตัวความแม่นยำควอด (Quad-Precision Floating-Point) 62
5.7 การเปรียบเทียบ RV32FD, ARM-32, MIPS-32 และ x86-32 โดยใช้ DAXPY 62
5.8 บทสรุป 63
5.9 เรียนรู้เพิ่มเติม 63

6 RV32A: คำสั่งอะตอมิก **70**
6.1 บทนำ 70
6.2 ข้อสังเกต สรุป 73
6.3 เรียนรู้เพิ่มเติม 73

7 RV32C: คำสั่งแบบบีบอัด **74**
7.1 บทนำ 74
7.2 การเปรียบเทียบ RV32GC, Thumb-2, microMIPS และ x86-32 76
7.3 ข้อสังเกตสรุป 77
7.4 เรียนรู้เพิ่มเติม 77

8 RV32V: เวกเตอร์ **82**
8.1 บทนำ 82
8.2 คำสั่งการคำนวณเวกเตอร์ 86
8.3 รีจิสเตอร์เวกเตอร์และการพิมพ์แบบไดนามิก 86
8.4 โหลดเวกเตอร์และบันทึกเวกเตอร์ 88
8.5 การทำงานแบบขนานในระหว่างการดำเนินการเวกเตอร์ 91
8.6 การดำเนินการแบบมีเงื่อนไขของการคำนวณเวกเตอร์ 91
8.7 การควบคุมความยาวเวกเตอร์ 92
8.8 ตัวอย่างเวกเตอร์ DAXPY ใน RV32V 93
8.9 คำสั่งขยายและคำสั่งแคบลง 95
8.10 Vector Register Move, Integer Extend, and Convert 97
8.11 Vector Instructions for Libraries 99
8.12 การเปรียบเทียบ RV64V กับ ARMv8 SVE และ x86-64 AVX SIMD . . . 107
8.13 ข้อสังเกตสรุป 114
8.14 เรียนรู้เพิ่มเติม 115

9 RV32B: การจัดการบิต **122**
9.1 บทนำ 122
9.2 Zba: การคำนวณที่อยู่หน่วยความจำ 123
9.3 Zbb: การจัดการบิตทั่วไป 123
9.4 Zbc: การคูณแบบไม่ใช้แคร์รี 124
9.5 Zbs: การจัดการบิตเดียว 124
9.6 ข้อสังเกตสรุป 124
9.7 เรียนรู้เพิ่มเติม 124

10 RV64: คำสั่งที่อยู่หน่วยความจำ 64 บิต **126**
10.1 บทนำ 126
10.2 การเปรียบเทียบกับ ISA 64 บิตอื่นโดยใช้การเรียงลำดับแบบแทรก 132
10.3 ขนาดโปรแกรม 134
10.4 ข้อสรุปท้ายบท 135
10.5 เรียนรู้เพิ่มเติม 136

11 RV32/64 สถาปัตยกรรมสิทธิพิเศษ **142**
11.1 บทนำ 142
11.2 โหมดเครื่องจักรสำหรับระบบฝังตัวอย่างง่าย 143
11.3 การจัดการข้อยกเว้นในโหมดเครื่องจักร 146
11.4 โหมดผู้ใช้และการแยกกระบวนการในระบบฝังตัว 151

11.5 โหมดหัวหน้างานสำหรับระบบปฏิบัติการสมัยใหม่ 153
11.6 ระบบหน่วยความจำเสมือนใน S-mode 155
11.7 CSR ระบุคุณสมบัติและประสิทธิภาพ 159
11.8 ข้อสังเกตสรุป 160
11.9 เรียนรู้เพิ่มเติม 160

12 ส่วนขยายเพิ่มเติม RISC-V ในอนาคต **164**
12.1 ส่วนขยายมาตรฐาน “B” สำหรับการจัดการบิต 164
12.2 ส่วนขยายมาตรฐาน “J” สำหรับภาษาที่แปลแบบไดนามิก 164
12.3 ส่วนขยายมาตรฐาน “P” สำหรับคำสั่ง Packed-SIMD 164
12.4 ส่วนขยายมาตรฐาน “Zk” สำหรับการเข้ารหัสลับแบบสเกลาร์ 164
12.5 ข้อสังเกตสรุป 165

A รายการคำสั่ง RISC-V **166**

B การแปลงคำสั่ง RISC-V **222**
B.1 บทนำ 222
B.2 การเปรียบเทียบ RV32I, ARM-32 และ x86-32 โดยใช้ Tree Sum . . . 224
B.3 บทสรุป 225

C การแปลภาษาไทย **230**
C.1 บทนำ 230
C.2 RISC-V in a Nutshell 230
C.3 RISC-V ง่ายเหมือน Pi 231
C.4 ทำไมต้องใช้ รหัสดำเนินการ (Opcode)? 231
C.5 การขยายบิตเครื่องหมาย 232
C.6 การจัดเรียงลำดับไบต์ 233
C.7 รายการตัวย่อ 234
C.8 ความสอดคล้องของผู้แปล 235
C.9 ข้อสรุปท้ายบท 245

ดัชนี **246**

สารบัญรูปภาพ

1.1 สมาชิกขององค์กรของ RISC-V International 3
1.2 การเติบโตของชุดคำสั่ง x86 ตลอดอายุการใช้งาน 4
1.3 คำอธิบายของคำสั่ง *ASCII Adjust after Addition*(`aaa`) ใน x86-32 . . 5
1.4 เวเฟอร์ขนาด 8 นิ้วของชิป RISC-V ที่ออกแบบโดย SiFive 7
1.5 ขนาดโปรแกรมสัมพัทธ์สำหรับ RV32G, ARM-32, x86-32, RV32C และ Thumb-2 11
1.6 จำนวนหน้าและจำนวนคำของคู่มือ ISA 15

2.1 แผนภาพของคำสั่ง RV32I 19
2.2 รูปแบบคำสั่ง RISC-V 19
2.3 แผนผังรหัสดำเนินการของ RV32I 20
2.4 รีจิสเตอร์ของ RV32I 23
2.5 การเรียงลำดับแบบแทรกในภาษา C 29
2.6 จำนวนคำสั่งและขนาดโค้ดสำหรับการเรียงลำดับแบบแทรก สำหรับแต่ละ ISA 30
2.7 บทเรียนที่นักออกแบบ RISC-V เรียนรู้จากข้อผิดพลาดของ ISA ในอดีต . 32
2.8 โค้ด RV32I สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5. 34
2.9 โค้ด ARM-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 35
2.10 โค้ด MIPS-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 36
2.11 โค้ด x86-32 สำหรับการเรียงลำดับแบบแทรกในรูปที่ 2.5 37

3.1 ขั้นตอนการแปลจากซอร์สโค้ดภาษา C ไปสู่โปรแกรมที่รันได้ 39
3.2 รหัสคำสั่งแอสเซมเบลอร์สำหรับรีจิสเตอร์จำนวนเต็มและรีจิสเตอร์ทศนิยมของ RISC-V 40
3.3 คำสั่งเทียม 32 คำสั่งของ RISC-V ที่ใช้รีจิสเตอร์ x0 ซึ่งเป็นรีจิสเตอร์ศูนย์ . 42
3.4 คำสั่งเทียม 28 คำสั่ง ของ RISC-V ที่ไม่ขึ้นกับรีจิสเตอร์ x0 ,รีจิสเตอร์ศูนย์ 43
3.5 โปรแกรม Hello World ภาษาซี (`hello.c`) 43
3.6 โปรแกรม Hello World ภาษาแอสเซมบลีใน RISC-V (`hello.s`) 44

3.7 โปรแกรม Hello World ภาษาเครื่องใน RISC-V (`hello.o`) 44
3.8 โปรแกรม Hello World ภาษาเครื่องใน RISC-V หลังการเชื่อมโยง 45
3.9 คำสั่งแอสเซมเบลอร์ทั่วไปของ RISC-V 46
3.10 การจัดสรรหน่วยความจำของ RV32I สำหรับโปรแกรมและข้อมูล 47

4.1 แผนภาพของคำสั่ง RV32M 52
4.2 แผนผังรหัสคำสั่งของ RV32M มีรูปแบบคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อคำสั่ง 53
4.3 รหัส RV32M หารด้วยค่าคงที่โดยการคูณ 53

5.1 แผนภาพของคำสั่ง Zfh, RV32F และ RV32D 57
5.2 รีจิสเตอร์จุดลอยตัวของ Zfh, RV32F, และ RV32D 58
5.3 รีจิสเตอร์ควบคุมและสถานะจุดลอยตัว 59
5.4 คำสั่งแปลงข้อมูลของ Zfh, RV32F และ RV32D 60
5.5 โปรแกรม DAXPY ซึ่งเน้นการคำนวณจำนวนจุดลอยตัว เขียนด้วยภาษา C 62
5.6 จำนวนคำสั่งและขนาดของโค้ดสำหรับ DAXPY ในสี่สถาปัตยกรรมชุดคำสั่ง 63
5.7 โค้ด RV32D สำหรับ DAXPY ในรูปที่ 5.5 65
5.8 โค้ด ARM-32 สำหรับ DAXPY ในรูปที่ 5.5 65
5.9 โค้ด MIPS-32 สำหรับ DAXPY ในรูปที่ 5.5 66
5.10 โค้ด x86-32 สำหรับ DAXPY ในรูปที่ 5.5 66
5.11 แผนผังรหัสดำเนินการของ RV32F มีรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง 67
5.12 แผนผังรหัสดำเนินการ ของ RV32D มีโครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง 68
5.13 แผนผังรหัสดำเนินการ Zfh มีรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง 69

6.1 ไดอะแกรมของคำสั่ง RV32A 71
6.2 แผนผังรหัสดำเนินการ RV32A มีโครงสร้างรหัสดำเนินการ, รูปแบบคำสั่ง ,ประเภทรูปแบบ และชื่อ 71
6.3 ตัวอย่างการซิงโครไนซ์สองแบบ 73

7.1 แผนภาพของคำสั่ง RV32C 75
7.2 คำสั่งและขนาดของโค้ดสำหรับ Insertion Sort และ DAXPY ใน ISA แบบบีบอัด 76
7.3 รหัส RV32C สำหรับการเรียงลำดับการแทรก ที่รูป 2.5 78
7.4 รหัส RV32DC สำหรับ DAXPY ที่รูป 2.5 78

7.5 แผนผัง รหัส ดำเนิน การ RV32C (bits[1 : 0] = 01) แสดง โครงสร้าง opcodes รูปแบบและชื่อ 79
7.6 แผนผัง รหัส ดำเนิน การ RV32C (bits[1 : 0] = 00) แสดง โครงสร้าง opcodes รูปแบบและชื่อ 79
7.7 แผนผัง รหัส ดำเนิน การ RV32C (bits[1 : 0] = 10) แสดง รายการ โครงสร้าง opcodes รูปแบบและชื่อ 80
7.8 รูปแบบคำสั่งแบบบีบอัด RVC 16 บิต 80

8.1 แผนภาพของคำสั่งเวกเตอร์ RV32V ที่ได้รับความนิยมมากที่สุด 84
8.2 รูปแบบรีจิสเตอร์เวกเตอร์ RV32V คือ vtype 88
8.3 ชื่อและความหมายของตัวบ่งชี้ความยาวและความกว้างของเวกเตอร์ . . . 89
8.4 โค้ด RV32V สำหรับ DAXPY ในรูปที่ 5.5 94
8.5 แผนภาพ ของ คำ สั่ง เลขคณิต และ ตรรกะ แบบ ขยาย และ ย่อ ขนาด ซึ่ง ไม่ แสดงในรูปที่ 8.1 96
8.6 แผนภาพของคำสั่งย้าย การขยายจำนวนเต็ม และการแปลงของ RV32V . 97
8.7 แผนภาพ ของ คำ สั่ง RV32V สำหรับ การ ลด, มา สก์, การ ประมวล ผล สัญญาณ ดิจิตอล, การ จัดการ องค์ ประกอบ และ การ คำนวณ เกิน ความ กว้างปกติที่มีบิตส่วนเกิน (carry) 100
8.8 แผนภาพของเวอร์ชันเต็มของคำสั่งถ่ายโอนข้อมูลซึ่งถูกละไว้จาก 8.1 . . 105
8.9 โค้ด x86-64 AVX2 สำหรับ DAXPY ในรูปที่ 5.5 108
8.10 โค้ด x86-64 AVX-512 สำหรับ DAXPY ใน รูปที่ 5.5 109
8.11 โค้ด ARMv8 SVE สำหรับ DAXPY ในรูปที่ 5.5 110
8.12 จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับ SIMD และ ISA แบบเวก เตอร์ 112
8.13 แผนผังรหัสดำเนินการ RV32V สำหรับการโหลดและการจัดเก็บหลัก . . 116
8.14 แผนผัง รหัส ดำเนิน การ RV32V สำหรับ การ ดำเนิน การ เลขคณิต และ ตรรกะบางส่วน 117
8.15 แผนผังรหัสดำเนินการ RV32V สำหรับการดำเนินการเลขคณิตบางส่วน . 118
8.16 รหัสดำเนินการ RV32V สำหรับการดำเนินการจุดทศนิยมบางส่วน 119
8.17 รหัสดำเนินการ RV32V สำหรับการดำเนินการจุดทศนิยมและมาสก์บางส่วน120

9.1 แผนผังคำสั่ง RV32B 122
9.2 แผนผังรหัสดำเนินการของ RV32B 125

10.1 แผนผังของชุดคำสั่ง RV64I 127
10.2 แผนผังของคำสั่ง RV64M และ RV64A 127
10.3 แผนผังของคำสั่ง Zfh, RV64F, และ RV64D 128
10.4 แผนผังของคำสั่ง RV64C 128

10.5 แผนผังรหัสดำเนินการ RV64 สำหรับคำสั่งพื้นฐานและส่วนขยายเสริม . . 129
10.6 แผนภาพ รหัสดำเนินการของส่วนขยายการจัดการบิตเสริมสำหรับ RV64 131
10.7 แผนภาพของคำสั่ง RV64B 131
10.8 จำนวนคำสั่งและขนาดโค้ดสำหรับการเรียงลำดับแบบแทรกในสี่ ISA . . . 133
10.9 ขนาดโปรแกรมสัมพัทธ์สำหรับ RV64G, ARM-64 และ x86-64 เทียบกับ RV64GC 134
10.10โค้ด RV64I สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 137
10.11โค้ด ARM-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 138
10.12โค้ด MIPS-64 สำหรับการเรียงลำดับการแทรกในรูปที่ 2.5 139
10.13โค้ด x86-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 140

11.1 ไดอะแกรมของคำสั่งสิทธิพิเศษของ RISC-V 142
11.2 รูปแบบคำสั่งสิทธิพิเศษของ RISC-V, รหัสดำเนินการ, ประเภทรูปแบบคำสั่ง และชื่อคำสั่ง 143
11.3 สาเหตุของข้อยกเว้นและขัดจังหวะใน RISC-V 144
11.4 `mstatus` CSR 146
11.5 รีจิสเตอร์ควบคุมและสถานะการขัดจังหวะของโหมดเครื่องจักร 146
11.6 รีจิสเตอร์สาเหตุของโหมดเครื่องจักร และโหมดหัวหน้างาน (`mcause` และ `scause`) 147
11.7 รีจิสเตอร์ที่อยู่ฐานของเวกเตอร์การดักจับในโหมดเครื่องจักรและโหมดหัวหน้างาน (`mtvec` และ `stvec`) 147
11.8 รีจิสเตอร์ CSR ที่เกี่ยวข้องกับข้อยกเว้นและการขัดจังหวะ 147
11.9 ระดับสิทธิพิเศษของ RISC-V และการเข้ารหัส 148
11.10โค้ด RISC-V สำหรับตัวจัดการการขัดจังหวะตัวจับเวลาแบบง่าย 150
11.11รีจิสเตอร์ที่อยู่และการกำหนดค่าของ PMP 152
11.12ครงสร้างของการกำหนดค่า PMP ในรีจิสเตอร์ `pmpcfg` 152
11.13รีจิสเตอร์การมอบหมายข้อยกเว้น 153
11.14รีจิสเตอร์การขัดจังหวะของโหมดหัวหน้างาน 154
11.15`sstatus` CSR 155
11.16รายการตารางหน้า RV32 Sv32 (PTE) 156
11.17รายการตารางหน้า RV64 Sv39 (PTE) 157
11.18รีจิสเตอร์ `satp` 157
11.19การเข้ารหัสของฟิลด์ MODE ในรีจิสเตอร์ควบคุมและสถานะ `satp` . . . 158
11.20ไดอะแกรมของกระบวนการแปลที่อยู่ Sv32 158
11.21CSR `misa` รายงาน ISA ที่รองรับ 161
11.22CSR `mvendorid` ให้รหัสผู้ผลิต JEDEC ของคอร์ 161
11.23CSR ระบุเครื่องจักร (`marchid`, `mimpid`, `mhartid`) 161

11.24CSR เวลาเครื่องจักร (`mtime` and `mtimecmp`) 161
11.25รีจิสเตอร์ควบคุมตัวนับ `mcounteren` และ `scounteren` 161
11.26CSR ตัวนับประสิทธิภาพฮาร์ดแวร์ 162
11.27ขั้นตอนการแปลที่อยู่เสมือนเป็นที่อยู่กายภาพ 162

B.1 คำสั่งการเข้าถึงหน่วยความจำ RV32I แปลงเป็น ARM-32 และ x86-32 . 222
B.2 คำสั่งเลขคณิต RV32I แปลงเป็น ARM-32 และ x86-32 223
B.3 คำสั่งการควบคุม RV32I แปลงเป็น ARM-32 และ x86-32 224
B.4 ฟังก์ชัน C ที่รวมค่าต่างๆ ในไบนารีทรี โดยใช้การสำรวจแบบตามลำดับ (in-order traversal) 226
B.5 โค้ด RV32I สำหรับการสำรวจทรีแบบตามลำดับ (in-order) 227
B.6 โค้ด ARM-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) 227
B.7 โค้ด x86-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) 228

C.1 โครงสร้างโค้ดโปรแกรมแอสเซมบลี 232

คำนำ

ยินดีต้อนรับ!

RISC-V ได้กลายเป็นปรากฏการณ์ที่ได้รับความนิยมอย่างรวดเร็ว ตั้งแต่เปิดตัวในปี 2010 เราเชื่อว่าคู่มือของโปรแกรมเมอร์ที่กระชับจะช่วยกระตุ้นความนิยมและส่งเสริมให้ผู้อ่านเข้าใจว่าทำไม RISC-V จึงเป็นชุดคำสั่งที่น่าสนใจ และแตกต่างจากสถาปัตยกรรมชุดคำสั่ง (ISA) แบบดั้งเดิมในอดีตอย่างไร

หนังสือสำหรับสถาปัตยกรรมชุดคำสั่งอื่นๆ เป็นแรงบันดาลให้เรา แม้ว่าเราจะหวังว่าความเรียบง่ายของ RISC-V จะทำให้เขียนได้สั้นกว่าหนังสือดีๆ อย่าง *See MIPS Run* ที่มีความยาวกว่า 500 หน้า ด้วยความยาวเพียงหนึ่งในสาม ของหนังสือเหล่านั้น อย่างน้อยในแง่นี้เราก็ประสบความสำเร็จ ในความเป็นจริง มี 10 บทที่แนะนำแต่ละส่วนประกอบของชุดคำสั่ง RISC-V แบบโมดูล จำนวน 170 หน้า—แม้ว่าจะมีภาพประกอบเฉลี่ยเกือบหนึ่งภาพต่อหน้า (รวม 10ภาพ)—ซึ่งทำให้อ่านได้รวดเร็ว

หลังจากอธิบายหลักการออกแบบชุดคำสั่งแล้ว เราได้แสดงให้เห็นว่าสถาปนิกของ RISC-V เรียนรู้จากชุดคำสั่งในช่วง 40 ปีที่ผ่านมา ด้วยการนำแนวคิดดี ๆ มาใช้และหลีกเลี่ยงข้อผิดพลาดในอดีต การประเมินสถาปัตยกรรมชุดคำสั่ง (ISA) นั้นขึ้นอยู่กับสิ่งที่ถูกตัดออกไม่น้อยไปกว่าสิ่งที่ถูกรวมไว้

จากนั้นเราแนะนำแต่ละส่วนประกอบของสถาปัตยกรรมแบบโมดูลนี้ผ่านลำดับของบทต่างๆ โดยทุกบทจะมีโปรแกรมในภาษาแอสเซมบลีของ RISC-V ที่แสดงการใช้คำสั่งที่แนะนำในบทนั้น ซึ่งช่วยให้โปรแกรมเมอร์ภาษาแอสเซมบลีเรียนรู้โค้ด RISC-V ได้ง่ายขึ้น นอกจากนี้ เรายังแสดงโปรแกรมที่เทียบเท่าใน ARM, MIPS และ x86 เพื่อเน้นย้ำถึงความเรียบง่ายและประโยชน์ด้านต้นทุน พลังงาน และประสิทธิภาพของ RISC-V

เพื่อให้หนังสือเล่มนี้อ่านสนุกขึ้น เราได้แทรกแถบข้างเกือบ 80 ข้อความไว้ที่ขอบหน้า พร้อมคำอธิบายที่น่าสนใจเกี่ยวกับเนื้อหา และยังมีภาพราว 120 ภาพที่ขอบหน้าเพื่อเน้นตัวอย่างการออกแบบ ISA ที่ดี (ขอบหน้าของเราใช้งานได้อย่างคุ้มค่า!) สำหรับผู้อ่านที่ทุ่มเท เราจะเพิ่มส่วนขยายความราว 35 ส่วนตลอดทั้งเล่ม ซึ่งผู้อ่านสามารถเจาะลึกในหัวข้อที่สนใจได้ ส่วนเหล่านี้ไม่จำเป็นต่อการทำความเข้าใจเนื้อหาอื่นในหนังสือ ดังนั้นหากไม่น่าสนใจก็สามารถข้ามไปได้ สำหรับผู้ที่หลงใหลในสถาปัตยกรรมคอมพิวเตอร์ เราได้

อ้างอิงบทความและหนังสือ 37 เล่มที่อาจขยายขอบเขตมุมมองของคุณ เราเองก็ได้เรียนรู้มากมายจากการอ่านงานเหล่านี้เพื่อเขียนหนังสือเล่มนี้!

ทำไมมีคำคมที่มีชื่อเสียงมากมาย?

เราเชื่อว่าคำคมช่วยให้การอ่านสนุกยิ่งขึ้น จึงได้สอดแทรกคำคม 34 ข้อไว้ทั่วทั้งเล่ม คำคมเหล่านี้เป็นกลไกที่มีประสิทธิภาพในการถ่ายทอดภูมิปัญญาจากรุ่นพี่สู่น้องใหม่ และช่วยกำหนดมาตรฐานวัฒนธรรมสำหรับการออกแบบ ISA ที่ดี นอกจากนี้ เรายังต้องการให้ผู้อ่านได้ซึมซับประวัติศาสตร์ของสาขานี้ด้วย จึงได้นำเสนอคำคมจากนักวิทยาการคอมพิวเตอร์และวิศวกรที่มีชื่อเสียงตลอดทั้งเล่ม

การแนะนำและการอ้างอิง

หนังสือเล่มบางนี้มีจุดมุ่งหมายให้เป็นทั้งบทนำและคู่มืออ้างอิงสำหรับ RISC-V สำหรับนักศึกษาและโปรแกรมเมอร์ระบบฝังตัวที่สนใจเขียนโค้ด RISC-V หนังสือเล่มนี้สมมติว่าผู้อ่านเคยเห็นชุดคำสั่งอย่างน้อยหนึ่งชุดมาก่อน หากยังไม่มีพื้นฐาน อาจลองอ่านหนังสือสถาปัตยกรรมเบื้องต้นที่เกี่ยวข้องของเรา ซึ่งใช้ RISC-V เป็นพื้นฐาน: *Computer Organization and Design RISC-V Edition, Second Edition: The Hardware Software Interface*

เอกสารอ้างอิงขนาดกะทัดรัดในหนังสือเล่มนี้ประกอบด้วย:

- **บัตรอ้างอิง**–เอกสารสรุปความยาวหนึ่งหน้า (สองด้าน) นี้ครอบคลุมทั้ง RV32GCV และ RV64GCV ซึ่งรวมถึงชุดคำสั่งพื้นฐานและส่วนขยายที่กำหนดไว้ทั้งหมด: RVI, RVM, RVA, RVF, RVD, RVC และแกนหลักของ RVV
- **แผนภาพคำสั่ง**–แผนภาพครึ่งหน้านี้แสดงคำอธิบายกราฟิกของแต่ละส่วนขยายคำสั่ง โดยเป็นภาพแรกของแต่ละบท แสดงรายชื่อเต็มของคำสั่ง RISC-V ในรูปแบบที่ช่วยให้เห็นความแตกต่างของแต่ละคำสั่งได้อย่างชัดเจน ดูภาพประกอบที่ 2.1, 4.1, 5.1, 6.1, 7.1, 8.1, 10.1, 10.2, 10.3, และ 10.4
- **ตารางรหัสคำสั่ง (Opcodes Maps)**–ตารางเหล่านี้แสดงโครงสร้างคำสั่ง รหัสคำสั่ง (opcode) รูปแบบคำสั่ง และชื่อย่อของคำสั่งสำหรับแต่ละส่วนขยายคำสั่ง โดยใช้พื้นที่เพียงบางส่วนของหน้า ดูภาพประกอบที่ 2.3, 3.3, 3.4, 4.2, 5.11, 5.12, 6.2, 7.6, 7.5, 7.7, 10.5, และ 11.1 (แผนภาพคำสั่งและตารางรหัสคำสั่งเป็นแรงบันดาลใจให้ใช้คำว่า atlas ในชื่อรองของหนังสือ)
- **รายการคำสั่ง**–ภาคผนวก A เป็นคำอธิบายอย่างละเอียดของทุกคำสั่งและคำสั่งเทียม (pseudoinstruction) ของ RISC-V รวมถึงชื่อการดำเนินการและตัวถูกดำเนินการ คำอธิบายเป็นภาษาไทย คำจำกัดความในภาษาการถ่ายโอนเรจิสเตอร์ ส่วน

ขยายของ RISC-V ที่เกี่ยวข้อง ชื่อเต็มของคำสั่ง รูปแบบคำสั่ง แผนภาพของคำสั่งที่แสดงรหัสคำสั่ง และการอ้างอิงถึงคำสั่งแบบบีบอัด ทั้งหมดนี้รวมอยู่ในพื้นที่ไม่ถึง 60 หน้า

- **การแปลงคำสั่ง**–ภาคผนวก B ช่วยโปรแกรมเมอร์ภาษาแอสเซมบลีที่มีประสบการณ์โดยให้ตารางที่แสดงคำสั่ง ARM-32 หรือ x86-32 ที่เทียบเท่ากับคำสั่ง RV32I รวมถึงแสดงผลลัพธ์ของคอมไพเลอร์ C สำหรับโปรแกรมการค้นหาแบบทรีอย่างง่าย ของสามสถาปัตยกรรมชุดคำสั่งนี้ และอธิบายความแตกต่างเล็กน้อยระหว่างสถาปัตยกรรมเหล่านี้ พร้อมคำแนะนำในการแปลโค้ดจากสถาปัตยกรรมเก่าไปเป็น RISC-V ซึ่งง่ายกว่าที่คิด
- **ตัวแปลภาษา**–ผู้อ่านที่ใช้ภาษาอื่นจะพบว่าพจนานุกรมของคำศัพท์ยากในภาคผนวก C ที่มีประโยชน์ขณะเรียนรู้ RISC-V ด้วยหนังสือเล่มนี้
- **ดัชนี**–ช่วยให้ผู้อ่านค้นหาหน้าที่อธิบายคำอธิบายคำสั่ง คำจำกัดความ หรือแผนภาพได้ทั้งจากชื่อเต็มหรือชื่อย่อ จัดเรียงเหมือนพจนานุกรม

ข้อผิดพลาดและเนื้อหาเพิ่มเติม

เราจะรวบรวมข้อผิดพลาดและเผยแพร่การอัปเดตปีละสองสามครั้ง เว็บไซต์ของหนังสือจะแสดงเวอร์ชันล่าสุดของหนังสือและคำอธิบายสั้นๆ เกี่ยวกับการเปลี่ยนแปลงจากเวอร์ชันก่อนหน้า สามารถดูรายการแก้ไขก่อนหน้า และรายงานรายการใหม่ได้ที่เว็บไซต์ของหนังสือ (www.riscbook.com) ขออภัยล่วงหน้าสำหรับปัญหาที่พบในฉบับนี้ และหวังว่าจะได้รับข้อเสนอแนะเพื่อปรับปรุงเนื้อหาของหนังสือเล่มนี้

ประวัติของฉบับพิมพ์ครั้งแรก

ที่การประชุมเชิงปฏิบัติการ RISC-V ครั้งที่ 6 ซึ่งจัดขึ้นระหว่างวันที่ 8–11 พฤษภาคม 2560 ที่เซี่ยงไฮ้ เราเห็นความจำเป็นสำหรับหนังสือเล่มนี้ และเริ่มเขียนไม่กี่สัปดาห์ต่อมา ด้วยประสบการณ์การเขียนหนังสือที่มากกว่าของแพตเตอร์สัน แผนคือให้เขาเขียนเนื้อหาส่วนใหญ่ของบทต่างๆ เราร่วมกันวางโครงสร้างและเป็นผู้ตรวจทานแรกสำหรับบทของกันและกัน โดยแพตเตอร์สันเป็นผู้เขียนบทที่ 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, การ์ดอ้างอิง, และคำนำ ส่วนวอเตอร์แมนเขียนบทที่ 11, ภาคผนวก A—ซึ่งเป็นส่วนที่ยาวที่สุดของหนังสือ—ภาคผนวก B, และเขียนโค้ดโปรแกรมทั้งหมดในหนังสือ วอเตอร์แมน ยังคงรักษารูปแบบ LaTeX ที่ได้รับจาก Armando Fox ซึ่งช่วยให้เราผลิตหนังสือเล่มนี้ได้

เราเสนอฉบับเบต้าของหนังสือให้กับนักศึกษา UC Berkeley 800 คน ในภาคเรียนฤดูใบไม้ร่วง 2560 ผู้อ่านพบเพียงข้อผิดพลาดด้านการพิมพ์และปัญหา LaTeX เพียงเล็กน้อย ซึ่งเราแก้ไขสำหรับฉบับพิมพ์ครั้งแรก นอกจากนี้ เรายังปรับปรุงไอคอนที่ขอบหน้าให้จำง่ายขึ้น และแก้ไขภาพประกอบบางภาพที่ดูไม่ดีเท่าที่หวังเมื่อพิมพ์ลงกระดาษ

ที่สำคัญ ฉบับพิมพ์ครั้งแรกได้ขยายบท 11 ให้รวมรีจิสเตอร์ควบคุมและสถานะกว่า 60 ตัว และเพิ่มภาคผนวก B เพื่อช่วยโปรแกรมเมอร์ที่สนใจแปลงโปรแกรมภาษาแอสเซมบลีจาก ISA รุ่นเก่าเป็น RISC-V

ฉบับแรกเผยแพร่ทันเวลาสำหรับการประชุมเชิงปฏิบัติการ RISC-V ครั้งที่ 7 ซึ่งจัดขึ้นในซิลิคอนแวลลีย์ ระหว่างวันที่ 28–30 พฤศจิกายน 2560

RISC-V เป็นผลพลอยได้จาก โครงการวิจัยของ Berkeley[1] ที่พัฒนาเทคโนโลยีเพื่อให้การสร้างฮาร์ดแวร์และซอฟต์แวร์แบบขนานง่ายขึ้น

ประกาศเกียรติคุณสำหรับฉบับแรก

เราขอขอบคุณ อาร์มันโด ฟอกซ์ สำหรับการใช้ระบบ LaTeX ของเขาและคำแนะนำในการเผยแพร่ด้วยตนเอง

คำขอบคุณอย่างยิ่งต่อผู้อ่านร่างต้นฉบับของหนังสือและให้คำแนะนำที่มีค่า: Krste Asanovi'c, Nikhil Athreya, C. Gordon Bell, Elkim Fuentes, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor และ Megan Wachs

และสุดท้าย เราขอบคุณนักศึกษา UC Berkeley จำนวนมากสำหรับความช่วยเหลือในการแก้ไขข้อบกพร่องและความสนใจอย่างต่อเนื่องในเนื้อหานี้!!

เราขอขอบคุณนักแปลจำนวนมาก ที่ได้แปลเนื้อหานี้ให้บริการฟรี ในภาษาจีน ญี่ปุ่น จีนกลาง โปรตุเกส สเปน และไทย (ดู riscbook.com)

ประวัติและคำขอบคุณสำหรับฉบับพิมพ์ครั้งที่สอง

เมื่อส่วนขยายเวคเตอร์เสร็จสมบูรณ์ เราเห็นว่าถึงเวลาเริ่มทำงานฉบับพิมพ์ครั้งที่สอง เนื่องจากบทเกี่ยวกับเวกเตอร์ในฉบับแรกเริ่มล้าสมัยอย่างมาก เราเริ่มติดต่อกันทางอีเมลในเดือนพฤษภาคม 2564 และเริ่มเขียนอย่างจริงจังในเดือนถัดมา นอกจากการปรับปรุงบทเวกเตอร์ (บทที่ 8) แล้ว เรายังตัดสินใจที่จะเพิ่มความแม่นยำครึ่งเท่าให้กับบทจุดทศนิยม (บทที่ 5) และเขียนบทใหม่เกี่ยวกับส่วนขยายการจัดการบิต (บทที่ 9) และไฮเปอร์ไวเซอร์ รวมถึงปรับปรุงบัตรอ้างอิง, คำนำ, บทนำ, บท RV64, บททิศทางในอนาคต และภาคผนวก A (วอร์เตอร์แมนเป็นผู้เขียนบทไฮเปอร์ไวเซอร์ ปรับปรุงภาคผนวก A และจัดหาตัวอย่างโปรแกรมทั้งหมด ส่วนแพตเตอร์สันรับผิดชอบการปรับปรุงที่เหลือ)

เรายินดีที่การเปลี่ยนแปลงเหล่านี้ เพิ่มความยาวของบทเพิ่มขึ้นเพียง 45 หน้า ในขณะที่ภาคผนวกเพิ่มขึ้น 38 หน้า ทำให้ฉบับพิมพ์ครั้งที่สองยังคงค่อนข้างบาง เราตัดสินใจให้ PDF ของเวอร์ชันภาษาอังกฤษของฉบับนี้เผยแพร่ฟรี เช่นเดียวกับรุ่นในภาษาอื่น ๆ (ดู riscbook.com)

เราขอขอบคุณผู้อ่านฉบับร่างของฉบับที่สอง สำหรับสำหรับข้อเสนอแนะที่มีค่า: Cliff Young และคนอื่น ๆ

เดวิด แพตเตอร์สัน และ แอนดรูว์ วอเตอร์แมน
๑๖ พฤศจิกายน พ.ศ. ๒๕๖๔
เบิร์กลีย์ แคลิฟอร์เนีย

หมายเหตุ

[1] http://parlab.eecs.berkeley.edu

1 ทำไมต้อง RISC-V ?

เลโอนาร์โด ดา วินชี (พ.ศ. 1995–2062) เป็นสถาปนิก ใน ยุค ฟื้นฟูศิลปวิทยา วิศวกร ประติมากรและจิตรกรภาพโมนาลิซา



ความเรียบง่ายคือความซับซ้อนขั้นสูงสุด

—เลโอนาโด ดา วินชี

1.1 บทนำ

เป้าหมายของ RISC-V (ออกเสียงว่า “RISC Five”) คือการเป็น *สถาปัตยกรรมชุดคำสั่ง* (Instruction Set Architecture หรือ *ISA*) ที่ใช้งานได้อย่างสากล:

- มันควรเหมาะสมกับโปรเซสเซอร์ทุกขนาด ตั้งแต่ตัวควบคุมฝังตัวขนาดเล็กที่สุด ไปจนถึงคอมพิวเตอร์สมรรถนะสูงที่เร็วที่สุ ด
- มันควรทำงานได้ดีกับซอฟต์แวร์สแตกและภาษาการเขียนโปรแกรมที่หลากหลายและเป็นที่นิยม
- มัน ควร รองรับ เทคโนโลยี การนำ ไป ใช้ งาน ทุก ประเภท: อาร์เรย์ เกต ที่ ตั้งโปรแกรมได้ (Field-Programmable Gate Arrays (FPGAs)), วงจรรวมเฉพาะงาน (Application-Specific Integrated Circuits (ASICs)), ชิปที่ออกแบบเต็มรูปแบบ (full-custom chips) และแม้แต่เทคโนโลยีอุปกรณ์ในอนาคต
- มันควรมีประสิทธิภาพสำหรับรูปแบบไมโครสถาปัตยกรรมทุกประเภท: การควบคุมแบบไมโครโค้ดหรือแบบฮาร์ดไวร์; ไปป์ไลน์แบบเรียงลำดับ (in-order), แยกส่วน (decoupled) หรือไม่เรียงลำดับ (out-of-order); การออกคำสั่งแบบเดี่ยวหรือซุปเปอร์สเกลาร์; และอื่นๆ
- มันควรสนับสนุนการปรับแต่งเฉพาะทางอย่างกว้างขวาง เพื่อเป็นฐานสำหรับตัวเร่งความเร็วที่กำหนดเอง ซึ่งมีความสำคัญเพิ่มขึ้นเมื่อกฎของมัวร์ค่อย ๆ ลดความสำคัญลง
- มันควรมีความมั่นคง โดยชุดคำสั่งพื้นฐาน (base ISA) ไม่ควรเปลี่ยนแปลง ที่สำคัญกว่านั้น ISA นี้จะต้อง ไม่ถูกยกเลิกเหมือนที่เกิดขึ้นกับ ISA ที่เป็นกรรมสิทธิ์ในอดีต เช่น AMD Am29000, Digital Alpha, Digital VAX, Hewlett Packard PA-RISC, Intel i860, Intel i960, Motorola 88000 และ Zilog Z8000

>$50B		>$5B, <$50B		>$0.5B, <$5B	
Alibaba	China	BAE Systems	UK	AMD	USA
Google	USA	MediaTek	Taiwan	Andes Technology	China
Huawei	China	Micron Tech.	USA	Cadence	USA
IBM	USA	Nvidia	USA	Integrated Device Tech.	USA
Samsung	Korea	NXP Semi.	Netherlands	Mellanox Technology	Israel
Sony	Japan	Qualcomm	USA	Seagate	USA
		Western Digital	USA	Xilinx	USA

รูปที่ 1.1: สมาชิกขององค์กรของ RISC-V International ณ เดือนสิงหาคม 2021 จัดอันดับตามยอดขายประจำปี บริษัทคอลัมน์ซ้าย: มียอดขายประจำปีเกิน $US 50B คอลัมน์กลาง: บริษัทที่มียอดขายน้อยกว่า $US 50B แต่มากกว่า $US 5B และบริษัทในคอลัมน์ขวามียอดขายน้อยกว่า $US 5B แต่มากกว่า $US 0.5B มูลนิธิรวมถึงองค์กรอีก 300 แห่งที่มีสำนักงานใหญ่ใน 28 ประเทศ ดูข้อมูลเพิ่มเติมได้ที่ www.riscv.org

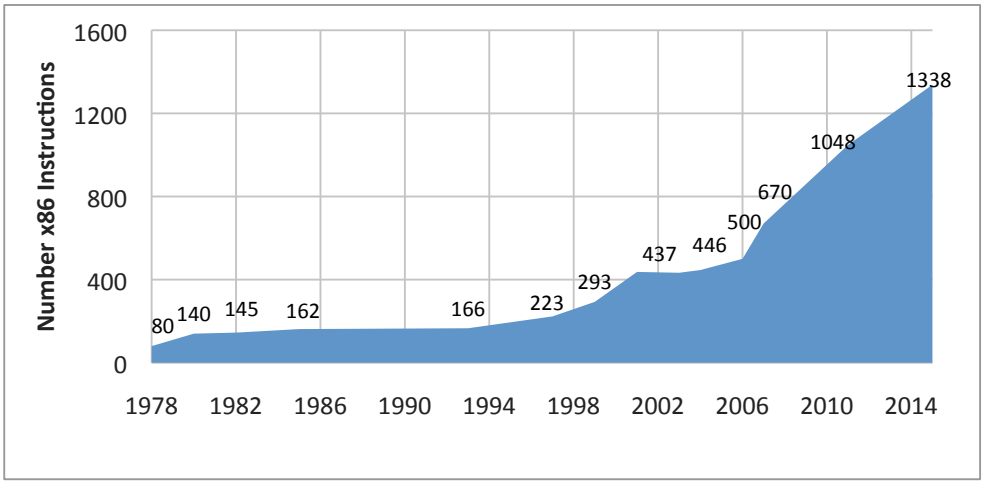
เราเพิ่มแถบด้านข้างที่ขอบหน้า เพื่อให้คำอธิบายที่น่าจะน่าสนใจ ตัวอย่างเช่น RISC-V เดิมถูกพัฒนาขึ้นเพื่อใช้ภายในในการวิจัยและหลักสูตรของ UC Berkeley มันกลายเป็น ISA แบบเปิดเพราะคนภายนอกเริ่มใช้งานมันด้วยตัวเอง สถาปนิกของ RISC-V ทราบถึงความสนใจจากภายนอกเมื่อพวกเขาเริ่มได้รับคำร้องเรียนเกี่ยวกับการเปลี่ยนแปลง ISA ในเอกสารการสอนของพวกเขา ซึ่งเผยแพร่บนเว็บ เฉพาะหลังจากที่สถาปนิกเข้าใจความต้องการนี้แล้ว พวกเขาถึงพยายามทำให้มันเป็นมาตรฐาน ISA แบบเปิด

RISC-V มีความไม่ธรรมดาไม่เพียงเพราะมันเป็น ISA (Instruction Set Architecture) ที่เกิดขึ้นใหม่—เริ่มต้นในปี 2010 ในขณะที่ ISA อื่นๆ ส่วนใหญ่มีอายุตั้งแต่ทศวรรษ 1970 หรือ 1980—แต่ยังเพราะมันเป็น ISA *แบบเปิด* ซึ่งต่างจากสถาปัตยกรรมเกือบทั้งหมดในอดีต อนาคตของมันไม่ได้ขึ้นอยู่กับชะตากรรมหรือความต้องการของบริษัทใดบริษัทหนึ่ง ซึ่งในอดีตเคยทำให้ ISA หลายตัวต้องล่มสลาย แต่มันเป็นของมูลนิธิที่ไม่แสวงหาผลกำไร และมีการออกแบบชุดคำสั่งที่เปิดเผยแทน เป้าหมายของมูลนิธิ RISC-V คือการรักษาความมั่นคงของ RISC-V พัฒนามันอย่างช้า ๆ และรอบคอบ ด้วยเหตุผลทางเทคนิคเพียงอย่างเดียว และพยายามทำให้มันได้รับความนิยมในด้านฮาร์ดแวร์เทียบเท่ากับที่ Linux ได้รับในด้านระบบปฏิบัติการ สัญญาณของความมีชีวิตชีวาของมันปรากฏในรูปที่ 1.1 ซึ่งแสดงรายชื่อสมาชิกองค์กรขนาดใหญ่ที่สุดของ RISC-V International

1.2 ISA แบบโมดูลาร์ เทียบกับ ISA แบบเพิ่มขยาย

> *อินเทลเดิมพันอนาคตของตนกับไมโครโปรเซสเซอร์ระดับสูง แต่กว่านั้นจะสำเร็จก็ยังต้องใช้เวลาอีกหลายปี เพื่อรับมือกับ Zilog อินเทลจึงได้พัฒนาโปรเซสเซอร์ชั่วคราวขึ้นมาและตั้งเรียกมันว่า 8086 โดยตั้งใจให้มันมีอายุสั้นและไม่มีรุ่นต่อมา แต่เรื่องราวกลับไม่เป็นเช่นนั้น โปรเซสเซอร์ระดับสูงที่วางแผนไว้กลับเข้าสู่ตลาดช้าเกินไป และเมื่อมันออกมา ก็ช้าเกินไปที่จะแข่งขันได้ ดังนั้นสถาปัตยกรรมของ 8086 จึงยังคงอยู่ต่อไป—มันพัฒนาไปเป็นโปรเซสเซอร์ 32 บิต และในที่สุดก็กลายเป็นโปรเซสเซอร์ 64 บิต ชื่อของมันเปลี่ยนไปเรื่อยๆ (80186, 80286, i386, i486, Pentium) แต่ชุดคำสั่งพื้นฐานยังคงเดิม*
>
> —สตีเฟน พี มอส, สถาปนิก ของ 8086 [Morse 2017]

แนวทางการออกแบบสถาปัตยกรรมคอมพิวเตอร์แบบดั้งเดิมคือ สถาปัตยกรรมชุดคำสั่ง (ISA) *แบบเพิ่มขยาย* ซึ่งโปรเซสเซอร์รุ่นใหม่ต้องนำไปใช้งานไม่เพียงแค่ส่วนขยาย ISA ใหม่เท่านั้น แต่ยังรวมถึงส่วนขยายทั้งหมดในอดีตด้วย วัตถุประสงค์คือ เพื่อรักษา*ความเข้ากันได้ย้อนหลังในระดับไบนารี* เพื่อให้โปรแกรมรุ่นเก่าที่เป็นไบนารีซึ่งมีอายุหลายสิบปีสามารถทำงานได้อย่างถูกต้องบนโปรเซสเซอร์รุ่นล่าสุด ข้อกำหนดนี้ เมื่อรวมกับความน่าสนใจทางการตลาด ในการประกาศคำสั่งใหม่พร้อมกับโปรเซสเซอร์รุ่นใหม่ ได้นำไปสู่ ISA ที่



รูปที่ 1.2: การเติบโตของชุดคำสั่ง x86 ตลอดอายุการใช้งาน x86 เริ่มต้นด้วยคำสั่ง 80 คำสั่งในปี 1978 และเติบโตขึ้น 16 เท่าเป็น 1,338 คำสั่งภายในปี 2015 และยังคงเติบโตต่อไปอย่างน่าทึ่ง กราฟนี้ยังนับว่าน้อยไป บล็อกของ Intel ระบุว่าจำนวนคำสั่งอยู่ที่ 3,600 คำสั่งในปี 2015 [Rodgers and Uhlig 2017] ซึ่งจะทำให้อัตราการเติบโตของ x86 กลายเป็นคำสั่งใหม่หนึ่งคำสั่ง *ทุกสี่วัน* ระหว่างปี 1978 ถึง 2015 เราได้นับคำสั่งในภาษาแอสเซมบลี ในขณะที่พวกเขาน่าจะนับคำสั่งในภาษาเครื่อง ตามที่อธิบายในบทที่ 8 ส่วนใหญ่ของการเติบโตของคำสั่งเกิดจาก ISA ของ x86 ที่พึ่งพาคำสั่ง SIMD เพื่อการประมวลผลแบบขนานระดับข้อมูล

```
AL เป็นรีจิสเตอร์ต้นทางและปลายทางค่าเริ่มต้น
If 4 บิตล่างของรีจิสเตอร์ AL มีค่า > 9
   หรือ กำหนดให้เฟลกตัวทด AF = 1
 Then
   บวก 6 เข้ากับ 4 บิตล่างของรีจิสเตอร์ AL และไม่ตรวจสอบ overflow
   เพิ่ม 1 ให้กับ 8 บิตบนของ AL
   กำหนดให้แฟลกการบวก CF = 1
   กำหนดให้เฟลกการบวกเพิ่มเติม AF = 1
 Else
   CF = AF = 0
4 บิตบนของรีจิสเตอร์ AL = 0
```

รูปที่ 1.3: คำอธิบายของคำสั่ง *ASCII Adjust after Addition*(`aaa`) ใน x86-32 คำสั่งนี้ดำเนินการคำนวณในรูปแบบ Binary Coded Decimal (BCD) งปัจจุบันกลายเป็นเทคโนโลยีที่ล้าสมัยและถูกลืมไปแล้ว x86 ยังมีคำสั่งที่เกี่ยวข้องอีกสามคำสั่งสำหรับการลบ (`aas`), การคูณ (`aam`) และการหาร (`aad`) โดยแต่ละคำสั่งใช้พื้นที่เพียงหนึ่งไบต์ ซึ่งทำให้ทั้งสี่คำสั่งนี้รวมกันแล้วใช้พื้นที่ 1.6% (4/256) ของพื้นที่ opcode อันมีค่าทั้งหมด

ขยายขนาดอย่างมากเมื่อเวลาผ่านไป ตัวอย่างเช่น รูปที่ 1.2 แสดงการเติบโตของจำนวนคำสั่งสำหรับ ISA ที่ครองตลาดในปัจจุบันอย่าง 80x86 ซึ่งเริ่มต้นในต่ปี 1978 แต่มีการเพิ่มคำสั่งประมาณ *สามคำสั่งต่อเดือน* ตลอดอายุการใช้งานที่ยาวนาน

แนวทางนี้หมายความว่าทุกการนำ x86-32 (ชื่อที่เราใช้สำหรับรุ่นที่อยู่ 32 บิตของ x86) ไปใช้งานจะต้องนำข้อผิดพลาดจากส่วนขยายในอดีตไปด้วย แม้ว่าจะไม่สมเหตุสมผลอีกต่อไป ตัวอย่างเช่น รูปที่ 1.3 อธิบายคำสั่ง ASCII Adjust after Addition (`aaa`) ของ x86 ซึ่งเป็นคำสั่งที่หมดประโยชน์ไปนานแล้ว

เพื่อให้เข้าใจง่าย ลองนึกภาพร้านอาหารที่เสิร์ฟเฉพาะเมนูอาหารราคาคงที่ ซึ่งเริ่มต้นจากมื้อเย็นเล็กๆ ที่มีแค่แฮมเบอร์เกอร์และมิลค์เชค เมื่อเวลาผ่านไป ร้านเพิ่มเฟรนช์ฟรายแล้วตามด้วยไอศกรีมซันเดย์ สลัด พาย ไวน์ พาสต้ามังสวิรัติ สเต๊ก เบียร์ และอื่นๆ เพิ่มไปเรื่อยๆ จนกลายเป็นงานเลี้ยงขนาดใหญ่ ภาพรวมอาจดูไม่สมเหตุสมผล แต่ลูกค้าสามารถหาอาหารที่เคยกินในอดีตได้ในร้านนี้ ข่าวร้ายคือ ลูกค้าจะต้องจ่ายค่าอาหารที่ราคาสูงขึ้นตามการขยายตัวของงานเลี้ยงในแต่ละมื้อ

นอกเหนือจากการเป็น ISA ที่ใหม่และเปิดกว้างแล้ว RISC-V ยังมีความไม่ธรรมดาเพราะมันเป็น ISA แบบ *โมดูลาร์* (modular) ซึ่งต่างจาก ISA ส่วนใหญ่ในอดีต แกนกลางคือชุดคำสั่งพื้นฐานที่เรียกว่า *RV32I* ซึ่งสามารถรันซอฟต์แวร์สแต็กเต็มรูปแบบได้ RV32I ถูกคงไว้และจะไม่เปลี่ยนแปลง ทำให้ผู้เขียนคอมไพเลอร์ นักพัฒนาระบบปฏิบัติการ และผู้เขียนโปรแกรมภาษาแอสเซมบลีมีเป้าหมายที่มั่นคง ความเป็นโมดูลาร์มาจากส่วนขยายมาตรฐานที่เป็นตัวเลือก ซึ่งฮาร์ดแวร์สามารถเลือกใช้หรือไม่รวมก็ได้ตามความต้องการของแอปพลิเคชัน ความเป็นโมดูลนี้ทำให้ RISC-V สามารถใช้งานในรูปแบบที่เล็กมากและใช้พลังงานต่ำ ซึ่งสำคัญสำหรับแอปพลิเคชันฝังตัว โดยการแจ้งให้คอมไพเลอร์ RISC-V ทราบว่าส่วนขยายใดรวมอยู่ มันจะสร้างโค้ดที่ดีที่สุดสำหรับฮาร์ดแวร์นั้น ตาม แนวปฏิบัติจะเติมตัวอักษรของส่วนขยายต่อท้ายชื่อเพื่อระบุว่ามีส่วนขยายใดบ้าง เช่น RV32IMFD รวมส่วน

หาก ซอฟต์แวร์ ใช้ คำสั่ง RISC-V ที่ ถูก ละเว้น จาก ส่วน ขยาย ที่ เป็น ตัวเลือก ฮาร์ดแวร์จะเกิดการดักจับ (trap) และดำเนินการฟังก์ชันที่ต้องการในซอฟต์แวร์ ซึ่งเป็นส่วนหนึ่งของไลบรารีมาตรฐาน

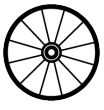
ขยายการคูณ (RV32M) จุดทศนิยมความแม่นยำเท่าเดียว (RV32F) และจุดทศนิยมความแม่นยำสองเท่า (RV32D) เข้ากับชุดคำสั่งพื้นฐานที่บังคับใช้ (RV32I)

เปรียบเทียบกลับไปที่ร้านอาหาร RISC-V เสนอเมนูแทนบุฟเฟ่ต์ พ่อครัวต้องปรุงเฉพาะสิ่งที่ลูกค้าต้องการ—ไม่ใช่งานเลี้ยงสำหรับทุกมื้อ—และลูกค้าจะจ่ายเฉพาะสิ่งที่สั่ง RISC-V ไม่จำเป็นต้องเพิ่มคำสั่งเพียงเพื่อความน่าสนใจทางการตลาด RISC-V International เป็นผู้ตัดสินใจว่าจะเพิ่มตัวเลือกใหม่ลงในเมนูเมื่อใด และจะทำเช่นนั้นเฉพาะด้วยเหตุผลทางเทคนิคที่มั่นคง หลังจากการอภิปรายอย่างเปิดเผยโดยคณะกรรมการผู้เชี่ยวชาญด้านฮาร์ดแวร์และซอฟต์แวร์ แม้ว่าจะมีตัวเลือกใหม่ในเมนู ตัวเลือกเหล่านั้นยังคงเป็นทางเลือก ไม่ใช่ข้อกำหนดใหม่สำหรับการใช้งานทั้งหมดในอนาคตเหมือนกับ ISA แบบเพิ่มขยาย

1.3 การออกแบบ ISA 101



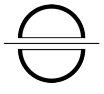
ค่าใช้จ่าย



ความเรียบง่าย



ประสิทธิภาพ



แยกสถาปัตย์ กับนำไปใช้



ที่ว่างสำหรับส่วนขยาย



ขนาดโปรแกรม



การเขียนโปรแกรม

ก่อนที่จะแนะนำ ISA ของ RISC-V การทำความเข้าใจหลักการพื้นฐานและการแลกเปลี่ยนที่สถาปนิกคอมพิวเตอร์ต้องพิจารณาขณะออกแบบ ISA จะเป็นประโยชน์ ด้านล่างคือรายการของเจ็ดตัวชี้วัด พร้อมด้วยไอคอน ที่เราจะใส่ในขอบหน้ากระดาษเพื่อเน้นกรณีที่ RISC-V ตอบสนองต่อตัวชี้วัดเหล่านี้ในบทต่อๆ ไป (ปกหลังของหนังสือฉบับพิมพ์มีคำอธิบายสำหรับไอคอนเหล่านี้)

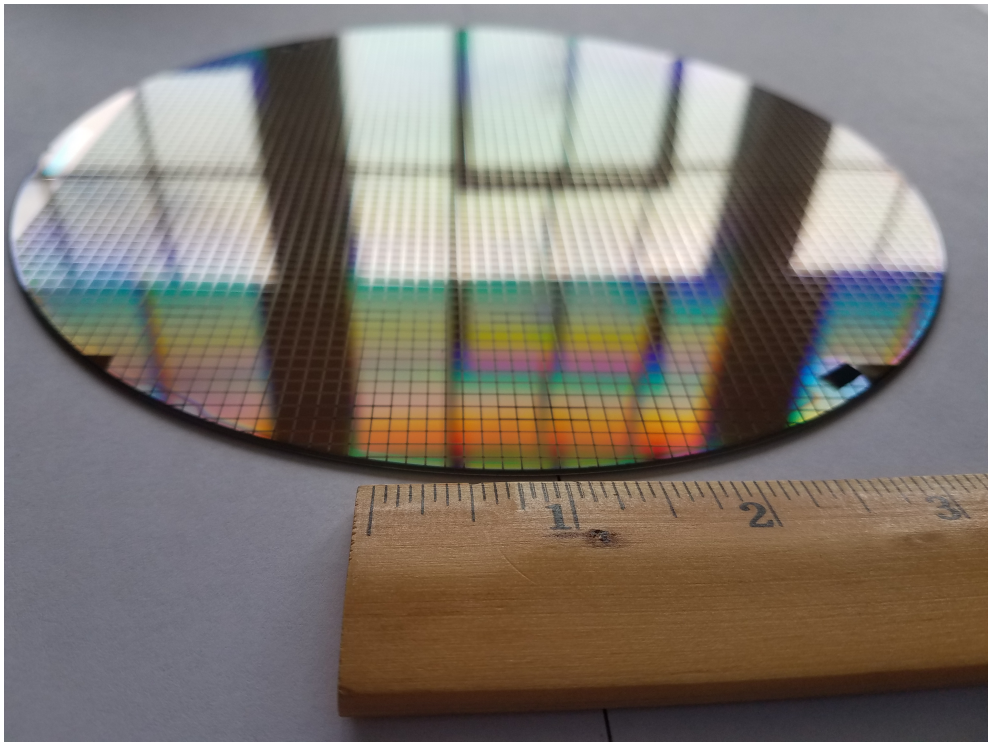
- ต้นทุน (ไอคอนเหรียญดอลลาร์สหรัฐ)
- ความเรียบง่าย (ไอคอนวงล้อ)
- ประสิทธิภาพ (ไอคอนมาตรวัดความเร็ว)
- การแยกสถาปัตยกรรมออกจากการนำไปใช้งาน (ไอคอนวงกลมที่แยกครึ่ง)
- พื้นที่สำหรับส่วนขยายเพิ่ม (ไอคอนหีบเพลง)
- ขนาดโปรแกรม (ไอคอนลูกศรสองตัวตรงข้ามบรรทัด)
- ความง่ายในการเขียนโปรแกรม / คอมไพล์ / ลิงก์ (“ไอคอน "ง่ายเหมือน A B C"”)

เพื่อแสดงให้เห็นว่าเราหมายถึงอะไร ในส่วนนี้ เราจะยกตัวอย่างตัวเลือกจาก ISA รุ่นเก่าที่ดูไม่ฉลาดเมื่อมองย้อนกลับไป และ RISC-V มักตัดสินใจได้ดีกว่ามาก

ต้นทุน โปรเซสเซอร์ถูกใช้งานในรูปแบบวงจรรวม ซึ่งมักเรียกว่า *ชิป* (chips) หรือ *ไดย์* (dies) มันถูกเรียกว่า“ไดย์” เพราะเริ่มต้นจากชิ้นส่วนของเวเฟอร์กลมเดียวที่ถูก*ตัด* (diced) ออกเป็นชิ้นเล็กๆ หลายชิ้น รูปที่ 1.4 แสดงเวเฟอร์ของโปรเซสเซอร์ RISC-V ต้นทุนขึ้นอยู่กับพื้นที่ของไดย์อย่างมาก:

$$cost \approx f(diev\ area^2)$$

เห็นได้ชัดว่า ไดย์ที่มีขนาดเล็กกว่าจะได้จำนวนไดย์ต่อเวเฟอร์มากขึ้น และต้นทุนส่วนใหญ่ของไดย์มาจากตัวเวเฟอร์ที่ผ่านการประมวลผลเอง สิ่งที่ไม่ชัดเจนคือ ไดที่เล็กกว่าจะมีอัตรา*ผลผลิต (yield)* สูงกว่า ซึ่งคือสัดส่วนของไดย์ที่ผลิตได้ และใช้งานได้ เหตุผลคือการ

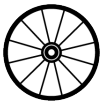


รูปที่ 1.4: เวเฟอร์ขนาด 8 นิ้วของชิป RISC-V ที่ออกแบบโดย SiFive เวเฟอร์ประกอบด้วยชิป RISC-V สองประเภท ที่ใช้สายการผลิตที่เก่ากว่าและมีขนาดใหญ่กว่า ชิป FE310 มีขนาด 2.65 มม×2.72 มม และชิปทดสอบของ SiFive มีขนาด 2.89 มม×2.72 มม เวเฟอร์นี้มีชิป FE310 จำนวน 1846 ชิ้น และชิปทดสอบ SiFive จำนวน 1866 ชิป รวมทั้งหมด 3712 ชิป

ผลิตซิลิกอนจะมีข้อบกพร่องเล็ก ๆ กระจายอยู่ทั่วเวเฟอร์ ดังนั้น ไดย์ที่เล็กกว่าจะมีสัดส่วนที่ได้รับผลกระทบจากข้อบกพร่องน้อยกว่า อย่างไรก็ตาม การเข้าใกล้ขีดจำกัดของความสามารถในการผลิตมากเกินไป การทำให้ไดย์เล็กเกินไป จะเพิ่มสัดส่วนของไดย์ที่เสียหายเนื่องจากการผลิตและการประกอบทำได้ยาก ซึ่งจะลดอัตราผลผลิตและเพิ่มต้นทุน

สถาปนิกต้องการรักษา ISA ให้เรียบง่ายเพื่อลดขนาดของโปรเซสเซอร์ที่นำไปใช้งาน ดังที่เราจะเห็นในบทต่อๆ ไป ISA ของ RISC-V เป็น ISA ที่เรียบง่ายกว่ามากเมื่อเทียบกับ ISA ของ ARM-32 ตัวอย่างที่ชัดเจนของผลกระทบจากความเรียบง่าย คือการเปรียบเทียบโปรเซสเซอร์ RISC-V Rocket กับโปรเซสเซอร์ ARM-32 Cortex-A5 ในเทคโนโลยีเดียวกัน (TSMC40GPLUS) โดยใช้แคชขนาดเท่ากัน (16 KiB) ไดย์ RISC-V มีขนาด 0.27 mm^2 เทียบกับ 0.53 mm^2 สำหรับ ARM-32 ซึ่งมีพื้นที่มากกว่าประมาณสองเท่า ไดย์ของ ARM-32 Cortex-A5 มีต้นทุนประมาณ 4 เท่า (2^2) เมื่อเทียบกับไดย์ของ RISC-V Rocket แม้แต่ไดย์ที่เล็กลงเพียง 10% ก็สามารถลดต้นทุนได้ถึง 1.2 เท่า (1.1^2)

โปรเซสเซอร์ ระดับ สูง สามารถเพิ่มประสิทธิภาพได้โดย การ รวม คำ สั่ง ที่เรียบ ง่าย เข้า ด้วย กัน โดย ไม่ เพิ่ม ภาระ ให้ กับ การนำ ไป ใช้ งาน ระดับ ล่าง ด้วย ISA ที่ใหญ่และซับซ้อนมาก ขึ้น เทคนิค นี้ เรียก ว่า มา โครฟิวชั่น (macrofusion) ซึ่ง รวม คำ สั่ง ขนาด ใหญ่ “ขนาดใหญ่” เข้า ด้วย กัน



ความเรียบง่าย

ความเรียบง่าย เมื่อพิจารณาถึงความไวต่อต้นทุนของความซับซ้อน สถาปนิกจึงต้องการ ISA ที่เรียบง่ายเพื่อลดพื้นที่ของไดย์ ความเรียบง่ายยังลดระยะเวลาในการออกแบบชิป และการตรวจสอบ ซึ่งอาจเป็นต้นทุนส่วนใหญ่ในการพัฒนาชิป ต้นทุนเหล่านี้ต้องถูกรวมเข้ากับต้นทุนของชิป โดยค่าใช้จ่ายส่วนเกินนี้จะขึ้นอยู่กับจำนวนชิปที่จัดส่ง ความเรียบง่ายยังช่วยลดต้นทุนของเอกสารประกอบ และลดความยากในการทำให้ลูกค้าเข้าใจวิธีการใช้ ISA

โปรเซสเซอร์ที่ เรียบ ง่าย สามารถ เป็น ประโยชน์ สำหรับ การใช้ งาน ฝังตัว (embedded applications) เนื่องจาก มัน ง่าย ต่อ การ คาด เดา เวลา การ ทำงาน โปรแกรมเมอร์ภาษา แอ สเซมบ ลี สำหรับ ไมโคร คอนโทรลเลอร์มักต้องการ รักษาการจับเวลาที่แม่นยำ ดังนั้นพวกเขาจึงพึ่งพาโค้ดที่ใช้จำนวนรอบนาฬิกาที่คาด เดาได้ ซึ่งสามารถนับได้ด้วย มือ

ด้านล่างนี้คือตัวอย่างที่ชัดเจนของความซับซ้อนของ ISA จาก ARM-32 :

```
ldmiaeq SP!, R4-R7, PC
```

คำสั่งที่เรียกว่า LoaD Multiple, Increment-Address on EQual ซึ่งหมายถึง การโหลดหลายครั้งและเพิ่มที่อยู่เมื่อเงื่อนไขเท่ากัน คำสั่งนี้ทำการโหลดข้อมูล 5 ครั้ง และเขียนลงใน 6 รีจิสเตอร์ แต่จะทำงานก็ต่อเมื่อรหัสเงื่อนไข EQ (Equal) ถูกตั้งค่าไว้เท่านั้น นอกจากนี้ยังเขียนผลลัพธ์ลงใน PC (Program Counter) ซึ่งหมายความว่ามันยังทำหน้าที่เป็นการกระโดดแบบมีเงื่อนไขอีกด้วย นับว่าเป็นคำสั่งที่ซับซ้อนมาก!

น่าแปลกคือ คำสั่งที่เรียบง่ายมีแนวโน้มจะถูกใช้งานมากกว่าคำสั่งที่ซับซ้อน เช่น x86-32 มีคำสั่ง `enter` ซึ่งถูกออกแบบมาให้เป็นคำสั่งแรก ที่ทำงานเมื่อเข้าสู่โปรซิเดอร์ เพื่อสร้าง stack frame สำหรับมัน (ดูบทที่ 3) แต่คอมไพเลอร์ส่วนใหญ่กลับใช้เพียงสองคำสั่ง x86-32 ที่เรียบง่ายนี้แทน:

```
push ebp        # ดันตัวชี้เฟรมไปที่สแต็ก
mov  ebp, esp # คัดลอกตัวชี้สแต็กไปยังตัวชี้เฟรม
```



ประสิทธิภาพ

ประสิทธิภาพ นอกจากชิปขนาดเล็กสำหรับงานฝังตัวแล้ว สถาปนิกมักกังวลเกี่ยวกับประสิทธิภาพ (performance) นอกเหนือจากต้นทุน ประสิทธิภาพสามารถแบ่งออกเป็น 3

องค์ประกอบ:

$$\frac{instructions}{program} \times \frac{average\ clock\ cycles}{instruction} \times \frac{time}{clock\ cycle} = \frac{time}{program}$$

แม้ว่า ISA ที่เรียบง่ายอาจต้องใช้คำสั่งมากกว่า ISA ที่ซับซ้อนในการรันโปรแกรม แต่ก็สามารถชดเชยได้โดยมีรอบนาฬิกาที่เร็วกว่า หรือใช้จำนวนรอบนาฬิกาต่อคำสั่ง (CPI: Clock Cycles Per Instruction) เฉลี่ยน้อยลง

ปัจจัยสุดท้ายคือผกผัน ของอัตรานาฬิกาดังนั้น อัตรานาฬิกา 1 GHz หมายถึงเวลาในแต่ละรอบสัญญาณนาฬิกา คือ 1 นาโนวินาที ($1/10^9$)

ตัวอย่างเช่น สำหรับเกณฑ์วัด CoreMark benchmark [Gal-On and Levy 2012] (100,000 รอบ) ประสิทธิภาพบน ARM-32 Cortex-A9 คือ

$$\frac{32.27\ B\ instructions}{program} \times \frac{0.79\ clock\ cycles}{instruction} \times \frac{0.71\ ns}{clock\ cycle} = \frac{18.15\ secs}{program}$$

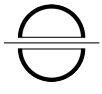
จำนวน รอบ สัญญาณนาฬิกา โดย เฉลี่ย อาจน้อย กว่า 1เนื่องจาก A9 และ BOOM [Celio et al. 2015] ที่เรียกว่าโปรเซสเซอร์ superscalar ซึ่งดำเนินการมากกว่าหนึ่งคำสั่งต่อรอบสัญญาณนาฬิกา

สำหรับการนำ RISC-V ไปใช้งานในรูปแบบ **BOOM** (**Berkeley Out-of-Order Machine**) สมการคือ

$$\frac{29.51\ B\ instructions}{program} \times \frac{0.72\ clock\ cycles}{instruction} \times \frac{0.67\ ns}{clock\ cycle} = \frac{14.26\ secs}{program}$$

โปรเซสเซอร์ ARM ไม่ได้ใช้คำสั่งน้อยกว่า RISC-V ในกรณีนี้ ดังที่เราจะเห็น คำสั่งที่เรียบง่ายมักเป็นคำสั่งที่ได้รับความนิยมมากที่สุด ดังนั้น ความเรียบง่ายของ ISA สามารถชนะในทุกตัวชี้วัด สำหรับโปรแกรมนี้ โปรเซสเซอร์ RISC-V มีประสิทธิภาพเพิ่มขึ้นเกือบ 10% ในแต่ละปัจจัยทั้งสาม ซึ่งส่งผลให้ได้เปรียบด้านประสิทธิภาพเกือบ 30% หาก ISA ที่เรียบง่ายยังส่งผลให้ได้ชิปที่มีขนาดเล็กลง ต้นทุนต่อประสิทธิภาพของมันก็จะยอดเยี่ยม

การ แยก สถาปัตยกรรม ออก จาก การนำ ไป ใช้ งาน ความแตกต่างดั้งเดิมระหว่าง *สถาปัตยกรรม* และ *การนำไปใช้งาน* ซึ่งย้อนกลับไปในทศวรรษ 1960 คือ สถาปัตยกรรมคือ สิ่งที่โปรแกรมเมอร์ภาษาเครื่องต้องรู้เพื่อเขียนโปรแกรมที่ถูกต้อง แต่ไม่เกี่ยวกับประสิทธิภาพของโปรแกรมนั้น ความท้าทายสำหรับสถาปนิกคือ การใส่คำสั่งใน ISA ที่ช่วยเพิ่มประสิทธิภาพหรือลดต้นทุนของการนำไปใช้งานหนึ่งในช่วงเวลาหนึ่ง แต่กลายเป็นภาระให้กับการนำไปใช้งานที่แตกต่างหรือในอนาคต



แยกสถาปัตย์ กับนำไปใช้

สำหรับ MIPS-32 ตัวอย่างที่น่าเสียใจคือ *การกระโดดแบบหน่วงเวลา* การกระโดดแบบมีเงื่อนไข สร้างปัญหาในการประมวลผลแบบไปป์ไลน์ เพราะเพราะโปรเซสเซอร์ต้องการมีคำสั่งถัดไปที่พร้อมประมวลผลในไปป์ไลน์ แต่ไม่สามารถตัดสินใจได้ว่าจะใช้คำสั่งถัดไปตามลำดับ (หากการกระโดดไม่เกิดขึ้น) หรือคำสั่งที่อยู่เป้าหมายของการกระโดด (หากการกระโดดเกิดขึ้น) สำหรับไมโครโปรเซสเซอร์ตัวแรก ของ MIPS มีไปป์ไลน์ 5 ขั้นตอน การเลือกคำสั่งนี้ นี้อาจทำให้ท่อส่งหยุดชะงักหนึ่งรอบสัญญาณนาฬิกา MIPS-32 แก้ปัญหานี้โดยกำหนดให้การกระโดดเกิดขึ้นในคำสั่ง *หลังจาก* คำสั่งถัดไป ดังนั้น คำสั่งที่ตามมาจะถูกประมวลผล*เสมอ* หน้าที่ของโปรแกรมเมอร์หรือผู้เขียนคอมไพเลอร์คือ ต้องใส่คำสั่งที่มีประโยชน์ลงใน *ช่องว่างหน่วงเวลา (delay slot)*

โปรเซสเซอร์ แบบ ไปป์ไลน์ในปัจจุบันคาดการณ์ผลลัพธ์ของการกระโดดโดยใช้ตัวทำนายแบบฮาร์ดแวร์ ซึ่งสามารถมีความแม่นยำเกิน 90% และทำงานได้กับไปป์ไลน์ที่มีความยาวใดก็ได้ โดยต้องมีกลไกในการล้างและเริ่มต้นไปป์ไลน์เมื่อมีการคาดการณ์ผิดพลาดเท่านั้น

แต่น่าเสียดายที่ “วิธีแก้ปัญหา” นี้ไม่ได้ช่วยโปรเซสเซอร์ MIPS-32 รุ่นหลังๆ ที่มีขั้นตอนไปป์ไลน์มากขึ้น (ซึ่งมีการดึงคำสั่งหลายคำสั่งก่อนที่ผลลัพธ์ของการกระโดดจะคำนวณ

ได้) แต่กลับทำให้นักเขียนโปรแกรม MIPS-32 นักพัฒนาคอมไพเลอร์ และนักออกแบบโปรเซสเซอร์ยากลำบากขึ้นนับแต่นั้นมา เนื่องจาก ISA แบบเพิ่มขยาย (incremental ISAs) ต้องการความเข้ากันได้ย้อนหลัง (ดูหัวข้อ 1.2) นอกจากนี้ ยังทำให้โค้ด MIPS-32 เข้าใจยากขึ้นมาก (ดูรูปที่ 2.10 หน้าที่ 36 บทที่ 2) ในขณะที่สถาปนิกไม่ควรใส่คุณลักษณะที่*ช่วย* การนำไปใช้งานเพียงแบบเดียวในช่วงเวลาหนึ่ง พวกเขาก็ไม่ควรวางคุณสมบัติที่ *ขัดขวาง*การนำไปใช้งานบางอย่างด้วย ตัวอย่างเช่น ARM-32 และ ISA บางตัวมีคำสั่ง **Load Multiple** ตามที่กล่าวถึงในหน้าที่แล้ว คำสั่งเหล่านี้สามารถเพิ่มประสิทธิภาพของการออกแบบไปป์ไลน์แบบคำสั่งเดียว แต่กลับลดประสิทธิภาพของการออกแบบไปป์ไลน์แบบแบบหลายคำสั่ง เหตุผลคือ การนำไปใช้งานแบบตรงไปตรงมาจะป้องกันการจัดตารางโหลดแต่ละครั้งของคำสั่ง **Load Multiple** ให้ทำงานคู่ขนานกับคำสั่งอื่น ๆ ซึ่งลดปริมาณงานของคำสั่งในโปรเซสเซอร์ดังกล่าว


ที่ว่างสำหรับส่วนขยาย

พื้นที่ว่างสำหรับส่วนขยายเพิ่ม ด้วยการสิ้นสุดของกฎของมัวร์ (Moore's Law) หนทางเดียวสำหรับการปรับปรุงต้นทุนต่อประสิทธิภาพ อย่างมีนัยสำคัญคือการเพิ่มคำสั่งที่กำหนดเอง สำหรับโดเมนเฉพาะ เช่น การเรียนรู้เชิงลึก (**deep learning**), เทคโนโลยีเสมือนผสานความจริง (AR), การเพิ่มประสิทธิภาพเชิงผสม (**combinatorial optimization**), กราฟิก และอื่นๆ นั่นหมายความว่าปัจจุบัน ISA ต้องสงวนพื้นที่ opcode สำหรับการปรับปรุงในอนาคต ในช่วงทศวรรษ 1970 และ 1980 เมื่อกฎของมัวร์ยังคงมีผลอย่างเต็มที่ มีการคำนึงถึงการสงวนพื้นที่ opcode สำหรับตัวเร่งความเร็ว (accelerators) ในอนาคตน้อยมาก สถาปนิกในสมัยนั้น ให้ความสำคัญกับการมีช่องที่อยู่ (address fields) และช่องค่าคงที่ (immediate fields) ที่ใหญ่ขึ้นเพื่อลดจำนวนคำสั่งที่ต้องประมวลผลต่อโปรแกรม ซึ่งเป็นปัจจัยแรกในสมการประสิทธิภาพ จากหน้าที่แล้ว

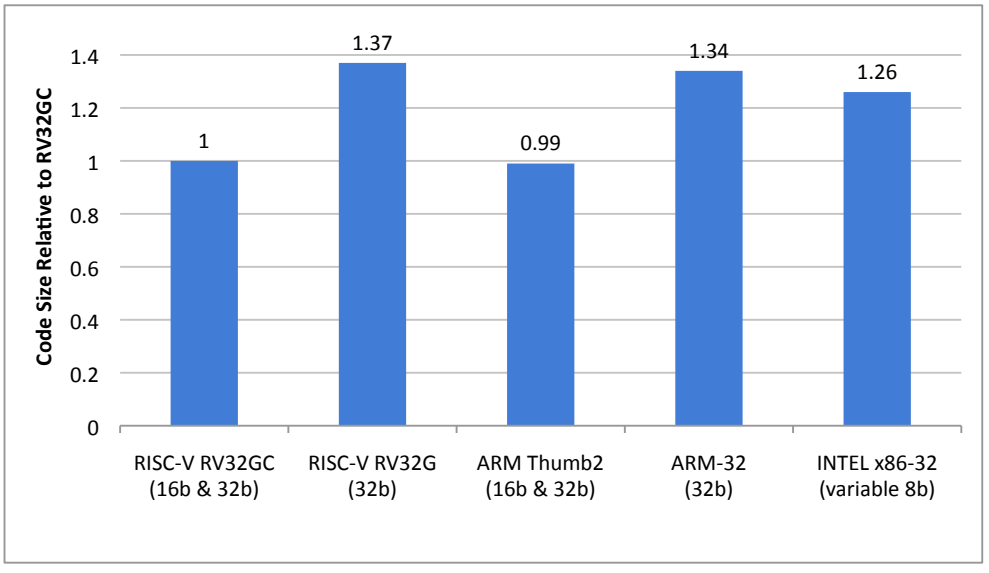
คำสั่ง ARM-32 `ldmiaeq` ที่กล่าว ถึง ข้าง ต้นนั้น ซับ ซ้อน ยิ่ง กว่า เพราะ เมื่อ มี การ ระ โดด ยัง สามารถเปลี่ยนโหมดชุดคำสั่งระหว่าง ARM-32 และ Thumb/Thumb-2 ได้ด้วย

ตัวอย่างของผลกระทบจากงการขาดแคลนพื้นที่ opcode คือเมื่อสถาปนิกของ ARM-32 พยายามลดขนาดโค้ดโดยเพิ่มคำสั่งขนาด 16 บิตลงใน ISA ที่เดิมมีคำขนาด 32 บิตเท่านั้น แต่กลับไม่มีพื้นที่opcode เหลือเพียงพอ ดังนั้น วิธีแก้ปัญหาเดียวคือการสร้าง ISA ใหม่ โดยเริ่มจากคำสั่งขนาด 16 บิต (Thumb) ต่อมาเป็น ISA ใหม่ที่มีทั้งคำสั่ง 16 บิต และ 32 บิต (Thumb-2) โดยใช้บิตโหมด (mode bit) เพื่อสลับระหว่าง ISA ของ ARM การเปลี่ยนโหมด ทำได้โดยนักเขียนโปรแกรมหรือคอมไพเลอร์กระโดดไปยังที่อยู่ไบต์ที่มี 1 ในบิตที่มีค่าน้อยที่สุด ซึ่งสามารถทำได้เพราะคำสั่ง 16 บิตและ 32 บิตควรมี 0 ในบิตนั้น


ขนาดโปรแกรม

ขนาดของโปรแกรม ยิ่งโปรแกรมมีขนาดเล็กลง พื้นที่บนชิปที่ต้องการสำหรับหน่วยความจำโปรแกรมก็ยิ่งน้อยลง ซึ่งเป็นต้นทุนที่สำคัญสำหรับอุปกรณ์ฝังตัว อันที่จริง ปัญหานี้เป็นแรงบันดาลใจให้สถาปนิกของ ARM เพิ่มคำสั่งที่สั้นลงใน ISA แบบ Thumb และ Thumb-2 ในภายหลัง โปรแกรมที่มีขนาดเล็กยังช่วยลดการพลาดในแคชคำสั่ง ซึ่งช่วยประหยัดพลังงาน เนื่องจากการเข้าถึง DRAM นอกชิปใช้พลังงานมากกว่าการเข้าถึง SRAM บนชิป และยังเพิ่มประสิทธิภาพด้วย ขนาดโค้ดที่เล็กสามารถเป็นหนึ่งในเป้าหมายของสถาปนิก ISA

ชุดคำสั่งของ x86-32 มีคำสั่งที่สั้นถึง 1 ไบต์ และยาวได้ถึง 15 ไบต์ ซึ่งคำสั่งที่มีความ



รูปที่ 1.5: ขนาดโปรแกรมสัมพัทธ์สำหรับ RV32G, ARM-32, x86-32, RV32C และ Thumb-2 สอง ISA สุดท้าย (RV32C, Thumb-2) มุ่งเน้นไปที่ขนาดโค้ดที่เล็ก โปรแกรมที่ใช้คือเกณฑ์วัด SPEC CPU2006 โดยใช้คอมไพเลอร์ GCC ข้อได้เปรียบด้านขนาดเล็กของ Thumb-2 เหนือ RV32C มาจากการลดขนาดโค้ดของคำสั่ง (**Load and Store Multiple**) เมื่อเข้าสู่โปรซีเจอร์ RV32C ไม่รวมคำสั่งเหล่านี้เพื่อรักษาการจับคู่แบบหนึ่งต่อหนึ่งกับคำสั่งของ RV32G ซึ่งไม่มีคำสั่ง **Load Multiple** และ **Store Multiple** เพื่อลดความซับซ้อนในการนำไปใช้งาน สำหรับโปรเซสเซอร์ระดับสูง (ดูด้านล่าง) บทที่ 7 อธิบาย RV32C ส่วน RV32G หมายถึง การรวมกันของส่วนขยายของ RISC-V (RV32M, RV32F, RV32D และ RV32A) ที่เรียกอย่างเป็นทางการว่า RV32IMAFD [Waterman 2016]

ตัวอย่าง คำ สั่ง x86-32 ขนาด 15 ไบต์ คำสั่ง `lock add dword ptr ds:[esi+ecx*4+0x12345678], 0xefcdab89` ถูก แอ สเซมเบิลเป็น (ในรูปแบบเลขฐานสิบหก): 67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef โดย 8 ไบต์สุดท้ายคือ 2 ตำแหน่งที่อยู่ และ 7 ไบต์แรก ระบุ การดำเนินการหน่วยความจำแบบอะตอมิก การบวก `add` ข้อมูล 32 บิต รีจิสเตอร์เซ็กเมนต์ข้อมูล `ds` รีจิสเตอร์ตำแหน่งที่อยู่ 2 ตัว `esi,ecx` และโหมดการกำหนดตำแหน่งที่อยู่แบบแบบสเกลอินเด็กซ์ `esi+ecx*4` ตัวอย่างคำสั่ง 1 ไบต์คือ คำสั่ง `inc eax` ที่ถูกแอสเซมเบิลเป็น 40 คำสั่งนี้มีที่อยู่ 4 ไบต์ สองที่อยู่และ 7 ไบต์เพื่อระบุการบวก (`add`), ข้อมูล 32 บิต (`dword ptr`), รีจิสเตอร์เซกเมนต์ข้อมูล (`ds`), รีจิสเตอร์ที่อยู่ (`esi, ecx`), โหมดการกำหนดที่อยู่แบบสเกลอินเด็กซ์ (`esi+ecx*4`), และ การดำเนินการหน่วยความจำแบบอะตอมิก (`lock`)

การเขียนโปรแกรม

ยาวแบบแปรผันเป็นไบต์ของ x86 ควรจะทำให้โปรแกรมมีขนาดเล็กลงกว่า ISA ที่จำกัดอยู่ที่คำสั่งยาว 32 บิต เช่น ARM-32 และ RISC-V และในทางตรรกะ คำสั่งที่มีความยาวแปรผัน 8 บิตก็น่าจะมีขนาดเล็กลงกว่า ISA ที่มีเฉพาะคำสั่ง 16 บิตและ 32 บิต เช่น Thumb-2 และ RISC-V ที่ใช้ส่วนขยาย RV32C (ดูบทที่ 7) รูปที่ 1.5 แสดงให้เห็นว่า ในขณะที่โค้ดของ ARM-32 และ RISC-V มีขนาดใหญ่กว่าโค้ดของ x86-32 ประมาณ 6% ถึง 9% เมื่อทุกคำสั่งยาว 32 บิต แต่ที่น่าประหลาดใจคือ x86-32 มีขนาดใหญ่กว่าเวอร์ชันที่ถูกบีบอัด (RV32C และThumb-2) ที่มีทั้งคำสั่ง 16 บิตและ 32 บิต ถึง 26%

หากออกแบบ ISA ใหม่โดยใช้คำสั่งขนาดแปรผัน 8 บิต อาจนำไปสู่โค้ดที่เล็กลงกว่า RV32C และ Thumb-2 แต่ในยุค 1970 สถาปนิกของ x86 รุ่นแรกมีข้อพิจารณาที่ต่างออกไป นอกจากนี้ ด้วยข้อกำหนดเรื่องความเข้ากันได้แบบย้อนหลังของ ISA แบบเพิ่มขยาย (หัวข้อ 1.2) คำสั่ง x86-32 ใหม่หลายร้อยคำสั่งมีขนาดยาวกว่าที่คาด เนื่องจากต้องใช้คำนำหน้า (prefix) หนึ่งหรือสองไบต์เพื่อบีบอัดลงในพื้นที่รหัสดำเนินการ (opcode) ที่ว่างจำกัดของ x86 เดิม

ความง่ายในการเขียนโปรแกรม การคอมไพล์ และการเชื่อมโยง เนื่องจากการเข้าถึงข้อมูลในเรจิสเตอร์นั้นเร็วกว่าการเข้าถึงข้อมูลในหน่วยความจำมาก การจัดสรรรีจิสเตอร์ที่ดีจึงสำคัญสำหรับคอมไพเลอร์ งานนี้จะง่ายกว่ามากเมื่อมีเรจิสเตอร์จำนวนมาก ในแง่นี้ ARM-32 มี 16 เรจิสเตอร์ ส่วน x86-32 มีเพียง 8 เรจิสเตอร์ ISA สมัยใหม่ส่วนใหญ่ รวมถึง RISC-V มีจำนวนเรจิสเตอร์จำนวนเต็มถึง 32 ตัว ซึ่งช่วยให้คอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลีทำงานง่ายขึ้น

อีกประเด็นหนึ่ง สำหรับคอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลี คือ การคำนวณความเร็วของลำดับโค้ด ดังที่เราจะได้เห็น คำสั่ง RISC-V มักใช้เวลาไม่เกินหนึ่งรอบนาฬิกาต่อคำสั่ง (หากไม่นับการพลาดแคช) ในขณะที่ ARM-32 และ x86-32 มีคำสั่งที่ใช้หลายรอบนาฬิกา แม้ว่าทุกอย่างจะอยู่ในแคช นอกจากนี้ แตกต่างจาก ARM-32 และ RISC-V คำสั่งการคำนวณ x86-32 สามารถมีตัวถูกดำเนินการในหน่วยความจำได้ แทนที่จะต้องอยู่ในเรจิสเตอร์ทั้งหมด คำสั่งที่ซับซ้อนและตัวถูกดำเนินการในหน่วยความจำ ทำให้ผู้ออกแบบโปรเซสเซอร์ยากต่อการคาดการณ์ประสิทธิภาพ

การสนับสนุน *โค้ดที่ไม่ขึ้นกับตำแหน่ง Position Independent Code (PIC)* มีประโยชน์สำหรับ ISA เพราะ มันสนับสนุนการเชื่อมโยงแบบไดนามิก (ดู หัวข้อ 3.5) เนื่องจากโค้ดไลบรารีที่แชร์สามารถอยู่ในที่อยู่ที่แตกต่างกันในโปรแกรมต่างๆ การกำหนดที่อยู่แบบสัมพัทธ์กับ PC (PC-relative) สำหรับการกระโดดแบบมีเงื่อนไขและการเข้าถึงข้อมูลเป็นประโยชน์อย่างมากสำหรับ PIC ในขณะที่ ISA เกือบทั้งหมดมีการกระโดดแบบสัมพัทธ์กับ PC แต่ x86-32 และ MIPS-32 กลับไม่มีการกำหนดที่อยู่ข้อมูลแบบสัมพัทธ์กับ PC

■ ***รายละเอียดเพิ่มเติม: ARM-32, MIPS-32 และ x86-32***

รายละเอียดเพิ่มเติม (Elaborations) เป็นส่วนเสริมที่ผู้อ่านสามารถเจาะลึกได้หากสนใจในหัวข้อใดนั้นๆ แต่ไม่จำเป็นต้องอ่านเพื่อเข้าใจเนื้อหาที่เหลือของหนังสือ ตัวอย่างเช่น ชื่อ ISA ที่เราใช้ไม่ใช่ชื่ออย่างเป็นทางการ ISA ที่ใช้ที่อยู่ 32 บิตของ ARM มีหลายรุ่น โดยรุ่นแรกเริ่มตั้งแต่ปี 1986 และรุ่นล่าสุดเรียกว่า ARMv7 ในปี 2005 โดยทั่วไป ARM-32 หมายถึง ISA ARMv7 ส่วน MIPS ก็มีรุ่น 32 บิตหลายรุ่น แต่เราหมายถึงรุ่นดั้งเดิมที่เรียกว่า MIPS I (ซึ่ง “MIPS32” เป็น ISA ที่แตกต่างและใหม่กว่า MIPS-32) สถาปัตยกรรมที่ใช้ที่อยู่ 16 บิตตัวแรกของ Intel คือ 8086 ในปี 1978 ซึ่ง ISA 80386 ขยายเป็นที่อยู่ 32 บิตในปี 1985 สัญลักษณ์ x86-32 ของเรามักหมายถึง IA-32 ซึ่งเป็นรุ่นที่ใช้ที่อยู่ 32 บิตของ x86 ด้วยความหลากหลายของ ISA เหล่านี้ เราพบว่าศัพท์ที่ไม่เป็นมาตรฐานของเรา ทำให้เกิดความสับสนน้อยที่สุด

1.4 ภาพรวมของหนังสือเล่มนี้

หนังสือเล่มนี้สันนิษฐานว่าคุณเคยเห็นชุดคำสั่งอื่นๆ มาก่อน RISC-V หากไม่เคย ควรดูหนังสือแนะนำสถาปัตยกรรมที่เกี่ยวข้องของเรา ซึ่งอ้างอิงจาก RISC-V [Patterson and Hennessy 2021]

บทที่ 2 แนะนำ RV32I ซึ่งเป็นชุดคำสั่งพื้นฐานจำนวนเต็มที่ถูกกำหนดให้คงที่ (frozen) และเป็นแกนหลักของ RISC-V บทที่ 3 อธิบายภาษาแอสเซมบลีของ RISC-V ส่วนที่เหลือ นอกเหนือจากที่แนะนำในบทที่ 2 รวมถึงข้อกำหนดการเรียกฟังก์ชัน และเทคนิคอันชาญฉลาดสำหรับการลิงก์ (linking) ภาษาแอสเซมบลีรวมถึงคำสั่ง RISC-V ที่ถูกต้องทั้งหมด รวมถึงคำสั่งที่มีประโยชน์บางคำสั่งที่อยู่นอกเหนือ RISC-V คำสั่งเทียม (pseudoinstructions) เหล่านี้ ซึ่งเป็นรูปแบบที่ชาญฉลาดของคำสั่งจริง ช่วยให้เขียนโปรแกรมภาษาแอสเซมบลีได้ง่ายขึ้นโดยไม่ต้องทำให้ ISA ซับซ้อน

สามบทถัดไปอธิบายส่วนขยายมาตรฐาน RISC-V ที่เมื่อรวมกับ RV32I แล้ว เรียกรวมกันว่า RV32G (G ย่อมาจาก general):

- บทที่ 4: การคูณและการหาร (RV32M)
- บทที่ 5: การดำเนินการจุดทศนิยม (RV32F และ RV32D)
- บทที่ 6: การดำเนินการหน่วยความจำอะตอมิก (RV32A)

“การ์ดอ้างอิง”ของ RISC-V หน้าที่ 3 และ 4 เป็นบทสรุปที่มีประโยชน์ของคำสั่ง RISC-V *ทั้งหมด*ในหนังสือเล่มนี้: RV32G, RV64G และ RV32/64V

บทที่ 7 อธิบายส่วนขยายเสริม RV32C ซึ่งเป็นตัวอย่างที่ยอดเยี่ยมของความสง่างามของ RISC-V โดยการจำกัดคำสั่ง 16 บิต ให้เป็นคำสั่งแบบย่อของคำสั่ง 32 บิตที่มีอยู่ใน RV32G ทำให้แทบไม่ต้องเพิ่มทรัพยากร แอสเซมเบลอร์สามารถเลือกขนาดคำสั่งได้ ทำให้โปรแกรมเมอร์ภาษาแอสเซมบลี และคอมไพเลอร์ไม่จำเป็นต้องรู้คำสั่ง RV32C ตัวถอดรหัสฮาร์ดแวร์สำหรับแปลคำสั่ง 16 บิตของ RV32C เป็นคำสั่ง 32 บิตของ RV32G ใช้เพียง 400 เกต ซึ่งเป็นเพียงไม่กี่เปอร์เซ็นต์ของการนำ RISC-V ไปใช้งานที่ง่ายที่สุด

การ์ดอ้างอิงนี้เรียกอีกชื่อว่า *กรีนการ์ด*เนื่องจากสีพื้นหลังของสรุป ISA แบบหนึ่งหน้าบนกระดาษแข็ง ในยุค 1960 เราเลือกพื้นหลังสีขาวเพื่อความชัดเจนในการอ่าน แทนที่จะใช้สีเขียวเพื่อความถูกต้องทางประวัติศาสตร์

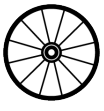
บทที่ 8 แนะนำ RV32V ซึ่งเป็นส่วนขยายเวกเตอร์ คำสั่งเวกเตอร์เป็นอีกตัวอย่างหนึ่งของความสง่างามของ ISA เมื่อเปรียบเทียบกับ *Single Instruction Multiple Data* (*SIMD*) จำนวนมากที่ใช้แนวทางแบบบังคับ (brute-force) ของ ARM-32, MIPS-32 และ x86-32 อันที่จริง คำสั่งนับร้อยที่เพิ่มใน x86-32 ในรูปที่ 1.2 เป็นคำสั่ง SIMD และยังมีคำสั่งเพิ่มเติมอีกนับร้อยที่กำลังจะมา RV32V เรียบง่ายกว่า ISA เวกเตอร์ส่วนใหญ่ เพราะมันเชื่อมโยงประเภทข้อมูลและความยาวกับรีจิสเตอร์เวกเตอร์ แทนที่จะฝังไว้ในรหัสดำเนินการ RV32V อาจเป็นเหตุผลที่น่าสนใจที่สุดในการเปลี่ยนจาก ISA ที่ใช้ SIMD แบบดั้งเดิมไปเป็น RISC-V

บทที่ 9 อธิบาย RV32B ซึ่งเป็นส่วนขยายการจัดการบิต คำสั่งเหล่านี้ช่วยในการคำนวณตำแหน่งที่อยู่ การดำเนินการตรรกะที่ไม่ค่อยพบ การหมุนบิต การนับบิต การตั้งค่าบิต และการดำเนินการที่ช่วยจัดการรหัสข้อผิดพลาดและการสร้างตัวเลขสุ่มเทียม

บทที่ 10 แสดงรุ่นที่ใช้ที่อยู่ 64 บิตของ RISC-V คือ RV64G ตามที่อธิบายในบทนี้ สถาปนิกของ RISC-V เพียงขยายขนาดรีจิสเตอร์และเพิ่มคำสั่งแบบ `word` `doubleword` หรือ `long` ของ RV32G ไม่กี่คำสั่งเพื่อขยายตำแหน่งที่อยู่จาก 32 บิตเป็น 64 บิต

บทที่ 11 อธิบายคำสั่งระบบ โดยแสดงวิธีที่ RISC-V จัดการการแบ่งหน้า (paging) และโหมดสิทธิพิเศษเครื่อง โหมดผู้ใช้ และโหมดผู้ควบคุม

บทสุดท้ายให้คำอธิบายสั้นๆ เกี่ยวกับส่วนขยายที่เหลือซึ่งปัจจุบันอยู่ระหว่างการพิจารณาโดย RISC-V International



ความเรียบง่าย

ส่วนที่ใหญ่ที่สุดของหนังสือคือภาคผนวก A ซึ่งเป็นสรุปชุดคำสั่งตามลำดับตัวอักษร โดยกำหนดสถาปัตยกรรมชุดคำสั่งของ RISC-V ทั้งหมดพร้อมส่วนขยายที่กล่าวถึงข้างต้น และคำสั่งเทียมทั้งหมด ในประมาณ 50 หน้า ซึ่งเป็นข้อพิสูจน์ถึงความเรียบง่ายของ RISC-V

ภาคผนวก B แสดงการดำเนินการภาษาแอสเซมบลีทั่วไปและคำสั่งที่สอดคล้องกันใน RV32I, ARM-32 และ x86-32 สามรูปนี้ตามด้วยโปรแกรม C ขนาดเล็กและผลลัพธ์ของคอมไพเลอร์สำหรับทั้งสาม ISA ภาคผนวกนี้มีสองวัตถุประสงค์: สำหรับผู้อ่านที่คุ้นเคยกับ ARM-32 หรือ x86-32 การการจับคู่ (mapping) เป็นอีกวิธีหนึ่งในการเรียนรู้ RISC-V วัตถุประสงค์ที่สอง คือช่วยโปรแกรมเมอร์ที่กำลังแปลโปรแกรมภาษาแอสเซมบลีที่มีอยู่ใน ISA รุ่นเก่าเป็น RISC-V

ภาคผนวก C ชี้ให้เห็นถึงประเด็นที่ท้าทายที่สุดในการทำความเข้าใจผ่านการแปล โดยหวังว่าผู้อ่านจะสามารถสัมผัสได้ถึงความสง่างามและความเรียบง่ายพื้นฐานเช่นเดียวกับที่หนังสือเล่มนี้เขียนขึ้นในต้นฉบับ คำศัพท์ที่พบได้บ่อยแต่ค่อนข้างเฉพาะเจาะจง เช่น รหัสดำเนินการ (opcode) & รหัสช่วยจำ (mnemonic), ลำดับไบต์ของระบบ (endian-ness), ช่องหน่วงเวลา (delay slot), การคำนวณแบบส่วนเติมเต็มฐานสอง (2's complement arithmetic), การขยายเครื่องหมายและการขยายศูนย์ของบิต (sign- & zero-extension) และการคำณวณแบบล้นหลักและขาดหลัก (overflow & underflow) ภาคผนวกประกอบด้วยอภิธานศัพท์ของตัวย่อ รวมถึงตารางคำศัพท์สำหรับการเรียนรู้ของผู้แปล เพื่อเพิ่มประโยชน์ให้กับดัชนีหลักที่ท้ายเล่ม สุดท้าย ผู้อ่านจะได้เห็นว่า RISC-V ส่งเสริมและ

จำนวนคำสั่ง (ISA)	จำนวนหน้า	จำนวนคำ	จำนวนชั่วโมง	จำนวนสัปดาห์
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	4898	2,186,259	182	4.5

รูปที่ 1.6: จำนวนหน้าและจำนวนคำของคู่มือ ISA [Waterman and Asanović 2017a], [Waterman and Asanović 2017b], [Intel Corporation 2016], [ARM Ltd. 2014] ที่ใช้ในการอ่านสมมติว่าอ่านที่ความเร็ว 200 คำต่อนาที เป็นเวลา 40 ชั่วโมงต่อสัปดาห์ อ้างอิงบางส่วนจากรูปที่ 1 ของ [Baumann 2017]

สนับสนุนสถาปัตยกรรมแบบเปิดและมาตรฐานอย่างไร ซึ่งกำลังได้รับความสนใจและความนิยมเพิ่มขึ้นในปัจจุบัน

เราปิดท้ายนังสือด้วยดัชนี

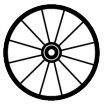
รายงานที่เขียนอย่างดีของ John von Neumannมีอิทธิพล อย่าง มาก จน รูป แบบ คอมพิวเตอร์ นี้ ถูก เรียกว่า สถาปัตยกรรม von Neumann แม้ว่า รายงาน นี้ จะ อิง จาก ผล งาน ของ ผู้ อื่น ก็ตาม รายงาน ดัง กล่าว ถูก เขียน ขึ้น สาม ปี ก่อน ที่ คอมพิวเตอร์ แบบ เก็บ โปรแกรม เครื่อง แรก จะ เริ่ม ใช้ งาน ได้!!

1.5 ข้อสรุปท้ายบท

> *จากวิธีการเชิงตรรกะอย่างเป็นทางการ สามารถเห็นได้ชัดเจนว่า มีชุดคำสั่งบางชุดที่ในทางนามธรรม เพียงพอต่อการควบคุมและทำให้เกิดการดำเนินการของลำดับการทำงานใดๆ ... อย่างไรก็ตาม ข้อพิจารณาที่สำคัญอย่างแท้จริงจากมุมมองในปัจจุบันในการเลือกชุดคำสั่งนั้น เป็นเรื่องในทางปฏิบัติมากกว่า: ความเรียบง่ายของอุปกรณ์ที่ชุดคำสั่งต้องการ ความชัดเจนในการประยุกต์ใช้กับปัญหาที่มีความสำคัญจริง ๆ และความเร็วในการจัดการกับปัญหาเหล่านั้น*
>
> —von Neumann et al. 1947 [Burks et al. 1946]

RISC-V เป็น ISA ล่าสุด ออกแบบใหม่ให้เรียบง่ายและเป็นสถาปัตยกรรมแบบเปิด โดยเรียนรู้จากข้อผิดพลาดของ ISA ในอดีต เป้าหมายของผู้ออกแบบ RISC-V คือให้มันมีประสิทธิภาพสำหรับอุปกรณ์คำนวณทุกประเภท ตั้งแต่อุปกรณ์ที่เล็กที่สุดไปจนถึงที่เร็วที่สุด ตามคำแนะของฟอนนอยมันน์เมื่อ 75 ปีก่อน ISA นี้เน้นความเรียบง่ายเพื่อลดต้นทุน พร้อมมีรีจิสเตอร์จำนวนมากและความเร็วของคำสั่งที่ชัดเจน เพื่อช่วยให้คอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลีสามารถแปลงปัญหาสำคัญให้เป็นโค้ดที่เหมาะสม และรวดเร็วได้

ข้อบ่งชี้ความซับซ้อนประการหนึ่งคือขนาดของเอกสาร รูปที่ 1.6 แสดงขนาดของคู่มือชุดคำสั่งสำหรับ RISC-V, ARM-32, และ x86-32 วัดจากจำนวนหน้าและคำ หากคุณอ่านคู่มือแบบเต็มเวลา—8 ชั่วโมงต่อวัน 5 วันต่อสัปดาห์—จะใช้เวลาครึ่งเดือนในการอ่านคู่มือ ARM-32 หนึ่งรอบ และหนึ่งเดือนเต็มสำหรับอ่านคู่มือ x86-32 ด้วยระดับความซับซ้อนนี้ อาจไม่มีใครเข้าใจ ARM-32 หรือ 86-32 ได้อย่างครบถ้วน ด้วยตัวชี้วัดที่สมเหตุสมผลนี้ RISC-V มีความซับซ้อนเพียง $\frac{1}{12}$ ของ ARM-32 และ $\frac{1}{10}$ ถึง $\frac{1}{30}$ ของ x86-32. อันที่จริงสรุปของสถาปัตยกรรมชุดคำสั่ง RISC-V รวมถึงส่วนขยายทั้งหมด ใช้เพียงสองหน้า (ดูบัตรอ้างอิง)



ความเรียบง่าย

ISA ที่เรียบง่ายและเปิดกว้างนี้ เริ่มต้นเมื่อวันที่ 18 พฤษภาคม 2010 และปัจจุบันได้รับการสนับสนุนจากมูลนิธิที่พัฒนาขึ้น โดยการเพิ่มส่วนขยายเสริมตามเหตุผลทางเทคนิคอย่างเคร่งครัด หลังจากการถกเถียงอย่างยาวนาน ความเปิดกว้างของ RISC-V ช่วยให้เกิด

เรา เฉลิม ฉลอง ครบ รอบ สิบปีของ RISC-Vด้วยการสัมภาษณ์ ผ่าน วิดีโอ 23 บุคคล จาก ทศวรรษ แรก ของ RISC-V เพื่อ แบ่ง ปัน ประสบการณ์ของ พวก เขา อธิบาย ว่า มัน ส่ง ผล ต่อ ชีวิต และ บริษัท ของ พวก เขา อย่างไร และ นำ เสนอ การ คาด การณ์ เกี่ยว กับ อนาคต ของ RISC-V ดูที่ riscv.org/risc-v-10th/

การใช้งานร่วมกันได้ฟรี ซึ่งช่วยลดต้นทุนและความเสี่ยงจากความลับที่เป็นอันตรายที่อาจซ่อนอยู่ในโปรเซสเซอร์

อย่างไร ก็ตาม ฮาร์ดแวร์ เพียง อย่าง เดียว ไม่ สามารถ สร้าง ระบบ ที่ สมบูรณ์ ได้ ค่า ใช้จ่าย ใน การ พัฒนา ซอฟต์แวร์ น่า จะ สูง กว่า ค่า ใช้ จ่าย ใน การ พัฒนา ฮาร์ดแวร์ มาก ดัง นั้น แม้ว่า ฮาร์ดแวร์ ที่ เสถียร จะ สำคัญ แต่ ซอฟต์แวร์ ที่ เสถียร นั้น สำคัญ ยิ่ง กว่า ระบบ ต้องการ ระบบ ปฏิบัติ การ ตัว โหลด บูต ซอฟต์แวร์ อ้างอิง และ เครื่องมือ ซอฟต์แวร์ ยอด นิยม มูลนิธิ RISC-V มอบ ความ เสถียร ให้ กับ ISA โดย รวม และ ชุด คำ สั่ง พื้น ฐาน ที่ ถูก กำหนด ให้ คง ที่ (frozen base) หมายความว่าแกน RV32I ซึ่งเป็นเป้าหมายของชุดซอฟต์แวร์จะไม่มีการเปลี่ยนแปลง ด้วยการยอมรับอย่างกว้างขวางและความเปิดกว้าง RISC-V สามารถท้าทายการครอบงำของ ISA ที่เป็นกรรมสิทธิ์ซึ่งมีอยู่ทั่วไปได้ คำว่า “สง่างาม” ไม่ค่อยถูกใช้กับสถาปัตยกรรมชุดคำสั่ง (ISA) แต่หลังจากอ่านหนังสือเล่มนี้ คุณอาจเห็นด้วยกับเราว่ามันเหมาะสมกับ RISC-V เราจะเน้นคุณสมบัติที่เราเชื่อว่า สะท้อนถึงถึงความสง่างามด้วยไอคอนโมนาลิซ่า ที่ขอบหน้า



สง่างาม

1.6 เรียนรู้เพิ่มเติม

ARM Ltd. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition, 2014. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/`.

A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132--137. ACM, 2017.

A. Burks, H. Goldstine, and J. Von Neumann. Preliminary discussion of the logical design of an electronic computer instrument. Technical report, 1946.

C. Celio, D. Patterson, and K. Asanovic. The Berkeley Out-of-Order Machine (BOOM): an industry-competitive, synthesizable, parameterized RISC-V processor. *Tech. Rep. UCB/EECS-2015--167, EECS Department, University of California, Berkeley*, 2015.

S. Gal-On and M. Levy. Exploring CoreMark - a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1--4*. September 2016.

S. P. Morse. The Intel 8086 chip and the future of microprocessor design. *Computer*, 50(4):8--9, 2017.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

S. Rodgers and R. Uhlig. X86: Approaching 40 and still going strong, 2017.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017a. URL `https://riscv.org/specifications/privileged-isa/`.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017b. URL `https://riscv.org/specifications/`.

2 RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V

ฟรานเซส เอลิซาเบธ “ฟราน” อัลเลน (พ.ศ.1995-2062) ได้รับรางวัลทัวริง จากผลงานของเธอในการพัฒนาคอมไพเลอร์ให้เหมาะสมที่สุด รางวัลทัวริงเป็นรางวัลสูงสุดในสาขาวิทยาการคอมพิวเตอร์ ซึ่ง บาง ครั้ง ถูก เรียก ว่า รางวัลโนเบลของวิทยาการคอมพิวเตอร์



...วิธีเดียวที่จะบรรลุเป้าหมายด้านประสิทธิภาพเป็นจริงและเข้าถึงได้สำหรับผู้ใช้ คือ การออกแบบคอมไพเลอร์และคอมพิวเตอร์ไปพร้อมกัน ด้วยวิธีนี้ คุณสมบัติต่างๆ จะไม่ถูกใส่เข้าไปในฮาร์ดแวร์ที่ซอฟต์แวร์ไม่สามารถใช้งานได้...

—ฟรานเซส เอลิซาเบธ “ฟราน” อัลเลน, 1981

2.1 บทนำ

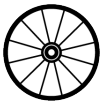
รูปที่ 2.1 เป็นการแสดงภาพกราฟิกหนึ่งหน้าของชุดคำสั่งพื้นฐาน RV32I ผู้อ่านสามารถดูชุดคำสั่ง RV32I ทั้งหมดได้โดยการต่อตัวอักษรที่ขีดเส้นใต้ จากซ้ายไปขวาสำหรับแต่ละแผนภาพ สัญกรณ์ชุดที่ใช้ { } จะระบุรูปแบบที่เป็นไปได้ของคำสั่ง โดยใช้ตัวอักษรที่ขีดเส้นใต้ หรือตัวอักษร _ ซึ่งหมายถึงไม่มีตัวอักษรสำหรับรูปแบบนี้ ยกตัวอย่างเช่น

$$\underline{\text{set}}\ \underline{\text{less}}\ \underline{\text{than}} \left\{ \begin{array}{l} _ \\ \underline{\text{i}}\text{mmediate} \end{array} \right\} \left\{ \begin{array}{l} _ \\ \underline{\text{u}}\text{nsigned} \end{array} \right\}$$

แสดงคำสั่ง RV32I สี่คำสั่งนี้: `slt`, `slti`, `sltu`, `sltiu`

จุดประสงค์ของแผนภาพเหล่านี้ ซึ่งจะเป็นภาพแรกของบทต่อไป คือ การให้ภาพรวมที่รวดเร็ว และเข้าใจง่ายของคำสั่งในแต่ละบท

2.2 รูปแบบคำสั่ง RV32I



ความเรียบง่าย

รูปที่ 2.2 แสดงรูปแบบคำสั่งพื้นฐานทั้งหกรูปแบบ: R-type สำหรับการดำเนินการระหว่างรีจิสเตอร์; I-type สำหรับค่าคงที่ขนาดสั้น (12 บิต) และการโหลด; S-type สำหรับการเก็บข้อมูล; B-type สำหรับการกระโดดแบบมีเงื่อนไข U-type สำหรับค่าคงที่ขนาดยาว (20 บิต) และ J-type สำหรับการกระโดดแบบไม่มีเงื่อนไข รูปที่ 2.3 แสดงรายการรหัสดำเนินการของคำสั่ง RV32I ในรูปที่ 2.1 โดยใช้รูปแบบจากรูปที่ 2.2

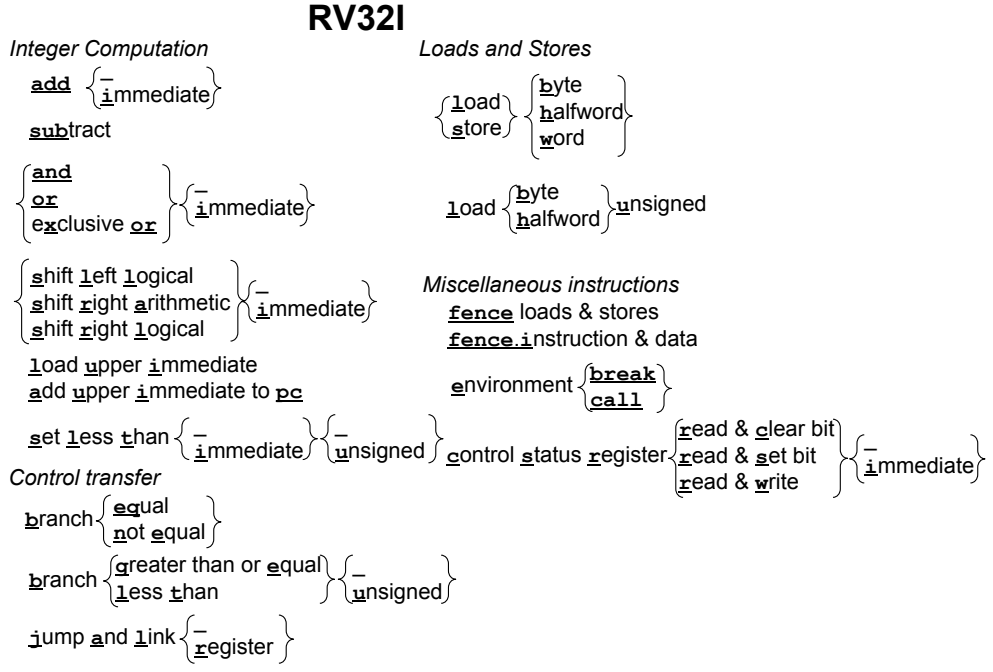


ค่าใช้จ่าย

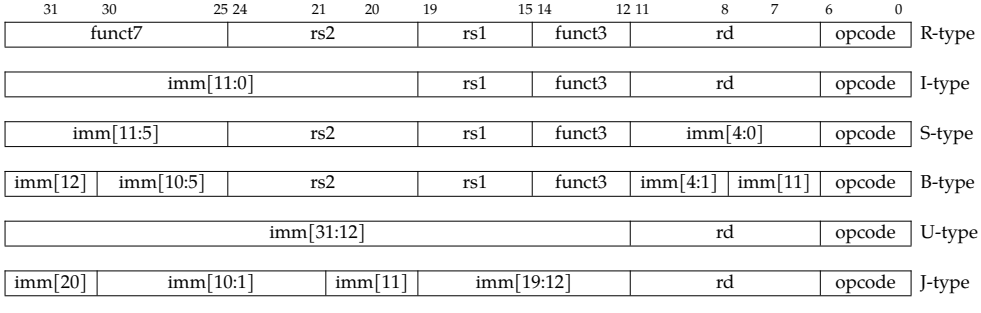


ประสิทธิภาพ

แม้แต่รูปแบบคำสั่งยังแสดงถึงตัวอย่างหลายประการที่แสดงให้เห็นว่า ที่สถาปัตยกรรมชุดคำสั่ง RISC-V ซึ่งมีความเรียบง่ายช่วยปรับปรุงประสิทธิภาพด้านต้นทุนได้อย่างไร ประการแรก มีเพียงหกประเภทรูปแบบคำสั่ง และทุกคำสั่งมีความยาว 32 บิต ซึ่งทำให้การถอดรหัสคำสั่งง่ายขึ้น ในขณะที่ ARM-32 และโดยเฉพาะ x86-32 มีรูปแบบคำสั่งจำนวน



รูปที่ 2.1: แผนภาพของคำสั่ง RV32I ตัวอักษรที่ขีดเส้นใต้จะถูกนำมาต่อกันจากซ้ายไปขวาเพื่อสร้างคำสั่งของ RV32I เครื่องหมายวงเล็บปีกกา { } หมายถึงแต่ละรายการในแนวตั้งภายในชุดเป็นรูปแบบที่แตกต่างกันของคำสั่งนั้นๆ เครื่องหมายขีดล่าง _ ภายในชุดหมายถึงตัวเลือกหนึ่งที่เป็นเพียงชื่อคำสั่ง จนถึงจุดนั้นโดยไม่มีตัวอักษรจากชุดนี้ ตัวอย่างเช่น สัญลักษณ์ใกล้มุมบนซ้ายแสดงถึงคำสั่งหกคำสั่งต่อไปนี้: `and`, `or`, `xor`, `andi`, `ori`, `xori`



รูปที่ 2.2: รูปแบบคำสั่ง RISC-V เราใส่ป้ายกำกับให้กับแต่ละส่วนย่อยของฟิลด์ค่าแบบทันที ด้วยตำแหน่งบิต (imm[*x*]) ในค่าทันทีที่ถูกสร้างขึ้น แทนที่จะใช้ตำแหน่งบิตในฟิลด์ค่าทันทีของคำสั่งตามที่มักทำกัน บทที่ 11 อธิบายว่าคำสั่งรีจิสเตอร์ควบคุมสถานะ ใช้รูปแบบ I-type ในลักษณะที่แตกต่างออกไปเล็กน้อย (รูปที่ 2.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

รูปที่ 2.3: แผนผังรหัสดำเนินการของ RV32I มีโครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ, และชื่อคำสั่ง (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

มาก ซึ่งทำให้การถอดรหัสมีค่าใช้จ่ายสูงในระบบที่มีต้นทุนต่ำ และเป็นความท้าทายด้านประสิทธิภาพ สำหรับการออกแบบโปรเซสเซอร์ระดับกลางและระดับสูง ประการที่สอง คำสั่งของ RISC-V รองรับตัวถูกดำเนินการจากรีจิสเตอร์สามตัว แทนที่จะใช้ฟิลด์เดียวกันสำหรับทั้งรีจิสเตอร์ต้นทางและปลายทาง อย่างใน x86-32 เมื่อการดำเนินการมีตัวถูกดำเนินการสามตัวที่แตกต่างกันโดยธรรมชาติ แต่ ISA มีเพียงคำสั่งแบบสองตัวถูกดำเนินการ คอมไพเลอร์หรือโปรแกรมเมอร์ภาษาแอสเซมบลีจะต้องใช้คำสั่งย้ายข้อมูล (move) เพิ่มเติมเพื่อรักษาค่าของตัวถูกดำเนินการปลายทาง ประการที่สาม ใน RISC-V ตัวระบุรีจิสเตอร์ที่ต้องอ่านและเขียนจะอยู่ในตำแหน่งเดียวกันในทุกคำสั่ง ซึ่งหมายความว่าการเข้าถึงรีจิสเตอร์สามารถเริ่มต้นได้ก่อนการถอดรหัสคำสั่ง ISA อื่นๆ จำนวนมาก (เช่น ARM-32 และ MIPS-32) ใช้ฟิลด์เดียวกันเป็นรีจิสเตอร์ต้นทางในบางคำสั่งและเป็นรีจิสเตอร์ปลายทางในคำสั่งอื่น ซึ่งบังคับให้ต้องเพิ่มฮาร์ดแวร์พิเศษในเส้นทางที่มีความสำคัญต่อเวลาเพื่อเลือกฟิลด์ที่เหมาะสม ประการที่สี่ ฟิลด์ค่าแบบทันที (immediate fields) ในรูปแบบเหล่านี้จะถูกขยายเครื่องหมาย (sign-extended) เสมอ และบิตเครื่องหมาย (sign bit) จะอยู่ในบิตที่มีนัยสำคัญสูงสุดของคำสั่งเสมอ การตัดสินใจนี้ทำให้การขยายเครื่องหมายค่าแบบทันที ซึ่งอาจอยู่ในเส้นทางที่มีความสำคัญต่อเวลา สามารถดำเนินการได้ก่อนการถอดรหัสคำสั่ง

การขยายเครื่องหมายของค่าแบบทันทีช่วยให้คำสั่งลอจิกทำงานได้ดีขึ้น ตัวอย่างเช่น การดำเนินการ x & 0xfffffff0 ใน RISC-V ใช้เพียงคำสั่งเดียว คือ `andi` แต่ใน MIPS-32 ต้องใช้สองคำสั่ง (`addiu` เพื่อโหลดค่าคงที่ จากนั้นดำเนินการระดับบิต `and`) เนื่องจาก MIPS ขยายศูนย์ (zero-extends) สำหรับค่าแบบทันทีของคำสั่งลอจิก ในขณะที่ ARM-32 ต้องเพิ่มคำสั่งพิเศษ `bic` ซึ่งทำการ rx & immediate เพื่อชดเชยการขยายศูนย์ของค่าแบบทันที

■ ***รายละเอียดเพิ่มเติม: รูปแบบ B และ รูปแบบ J คืออะไร?***

ตามที่กล่าวไว้ด้านล่าง ฟิลด์ค่าคงที่ทันที (imm) ถูกหมุน 1 บิต สำหรับคำสั่งกระโดดแบบมีเงื่อนไข ซึ่งเป็นรูปแบบที่เปลี่ยนแปลงจากรูปแบบ S ที่เรียกชื่อใหม่ว่ารูปแบบ B ฟิลด์ค่าคงที่ทันทีถูกหมุนเช่นกันสำหรับคำสั่งกระโดด ซึ่งเป็นรูปแบบที่เปลี่ยนแปลงจากรูปแบบ U ที่เรียกใหม่ว่ารูปแบบ J จริงๆ แล้วมีรูปแบบพื้นฐานสี่รูปแบบ แต่เราสามารถนับ RISC-V ด้วยเหตุผลในการออกแบบคำสั่งว่ามีหกรูปแบบได้

เพื่อช่วยโปรแกรมเมอร์ รูปแบบบิตที่เป็นศูนย์ทั้งหมด ถือเป็นคำสั่ง ที่ไม่ถูกต้องใน RV32I ดังนั้น การกระโดดผิดพลาดไปยังพื้นที่หน่วยความจำที่เป็นศูนย์ จะทำให้เกิดการดักจับ (trap) ทันที ซึ่งช่วยในการดีบัก ในทำนองเดียวกัน รูปแบบบิตที่เป็นหนึ่งทั้งหมดก็เป็นคำสั่งที่ไม่ถูกต้อง ซึ่งจะดักจับข้อผิดพลาดทั่วไปอื่นๆ เช่น อุปกรณ์หน่วยความจำแบบไม่ลบเลือนที่ยังไม่ได้ตั้งโปรแกรม บัสหน่วยความจำที่ไม่ได้เชื่อมต่อ หรือชิปหน่วยความจำที่เสียหาย


การเขียนโปรแกรม

เพื่อให้มีพื้นที่เพียงพอสำหรับการขยาย ISA ชุดคำสั่งพื้นฐาน RV32I ใช้พื้นที่การเข้ารหัสน้อยกว่า ⅛-th ของคำสั่งขนาด 32 บิต นักออกแบบยังเลือก opcode ของ RV32Iอย่างระมัดระวังเพื่อให้คำสั่งที่มีการดำเนินการในเส้นทางข้อมูล (datapath) ร่วมกันใช้ค่าเดียวกันในบิตของ opcode ให้มากที่สุดเท่าที่จะเป็นไปได้ ซึ่งช่วยลดความซับซ้อนของวงจรควบคุม ในท้ายที่สุด ดังที่เราจะเห็น ตำแหน่งที่อยู่ของการกระโดดแบบมีเงื่อนไข และการกระโดดแบบไม่มีเงื่อนไขในรูปแบบ B และ J จะต้องเลื่อนซ้าย 1 บิต เพื่อคูณตำแหน่งที่อยู่ด้วย 2 ซึ่งทำให้การกระโดดและการกระโดดแบบมีเงื่อนไข มีระยะที่กว้างขึ้น RISC-V จะหมุนบิตในตัวถูกดำเนินการแบบทันที จากตำแหน่งที่เป็นธรรมชาติเพื่อลดการกระจายสัญญาณของคำสั่ง และต้นทุนการการเลือกค่าแบบทันทีลงเกือบครึ่งหนึ่ง ซึ่งช่วยลดความ


ที่ว่างสำหรับส่วนขยาย


ค่าใช้จ่าย

การใช้งาน RISC-V ทั้งหมดใช้ opcode เดียวกันสำหรับส่วนขยายที่เป็นตัวเลือก เช่น RV32M, RV32F และอื่นๆ ส่วนขยายที่ไม่ใช่มาตรฐาน ซึ่งมีลักษณะเฉพาะสำหรับโปรเซสเซอร์ จะถูกจำกัดไว้ในพื้นที่ opcode ที่สงวนไว้ใน RISC-V

ซับซ้อนของวงจรเส้นทางข้อมูลในระบบที่มีต้นทุนต่ำ

มีอะไรแตกต่าง? เราจะจบแต่ละส่วนของบทนี้ และบทต่อๆ ไป ด้วยคำอธิบายว่า RISC-V แตกต่างจาก ISA อื่นๆ อย่างไร ความแตกต่างมักอยู่ที่สิ่งที่ RISC-V ไม่มี นักออกแบบแสดงให้เห็นถึงรสนิยมที่ดี ผ่านฟีเจอร์ที่พวกเขาเลือกตัดออก เช่นเดียวกับสิ่งที่พวกเขาเลือกใส่เข้ามา

ฟิลด์ค่าแบบทันทีขนาด 12 บิตของ ARM-32 ไม่ได้เป็นเพียงค่าคงที่ แต่เป็นอินพุตสำหรับฟังก์ชันที่สร้างค่าคงที่: 8 บิตจะถูกขยายศูนย์ (zero-extended) ให้เต็มความกว้าง จากนั้นหมุนขวาด้วยค่าที่ได้จาก 4 บิตที่เหลือคูณด้วย 2 ความหวังคือ การเข้ารหัสค่าคงที่ที่มีประโยชน์มากขึ้นใน 12 บิต จะช่วยลดจำนวนคำสั่งที่ต้องดำเนินการ ARM-32 ยังใช้ 4 บิตที่มีค่าของรูปแบบคำสั่งส่วนใหญ่สำหรับการดำเนินการแบบมีเงื่อนไข แม้ว่าการดำเนินการแบบมีเงื่อนไขจะถูกใช้ไม่บ่อย แต่ก็เพิ่มความซับซ้อนให้กับโปรเซสเซอร์แบบนอกลำดับ (*out-of-order processors*)



ประสิทธิภาพ

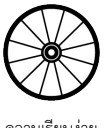
> ■ ***รายละเอียดเพิ่มเติม: โปรเซสเซอร์แบบนอกลำดับ***
>
> เป็นโปรเซสเซอร์ความเร็วสูงที่ใช้การประมวลผลแบบท่อ (pipelined) ซึ่งดำเนินการคำสั่งอย่างฉวยโอกาส (opportunistically) แทนการดำเนินการตามลำดับโปรแกรมแบบล็อกสเต็ป (lock-step) คุณสมบัติที่สำคัญของโปรเซสเซอร์ประเภทนี้คือ *การเปลี่ยนชื่อรีจิสเตอร์* ซึ่งจับคู่ชื่อรีจิสเตอร์ในโปรแกรมกับรีจิสเตอร์ทางกายภาพภายในที่มีจำนวนมากกว่า ปัญหากับการดำเนินการแบบมีเงื่อนไขคือ รีจิสเตอร์ทางกายภาพใหม่ต้องถูกเขียนข้อมูลไม่ว่าสภาพเงื่อนไขจะเป็นจริงหรือไม่ ดังนั้น ค่าปัจจุบันของรีจิสเตอร์ปลายทางต้องถูกอ่านเป็นตัวถูกดำเนินการที่สาม เพื่อคัดลอกไปยังรีจิสเตอร์ปลายทางใหม่ในกรณีที่เงื่อนไขไม่เป็นจริง ตัวดำเนินการพิเศษนี้เพิ่มต้นทุนของไฟล์รีจิสเตอร์ ตัวเปลี่ยนชื่อรีจิสเตอร์ และฮาร์ดแวร์สำหรับการดำเนินการแบบนอกลำดับ

2.3 รีจิสเตอร์ RV32I



การเขียนโปรแกรม

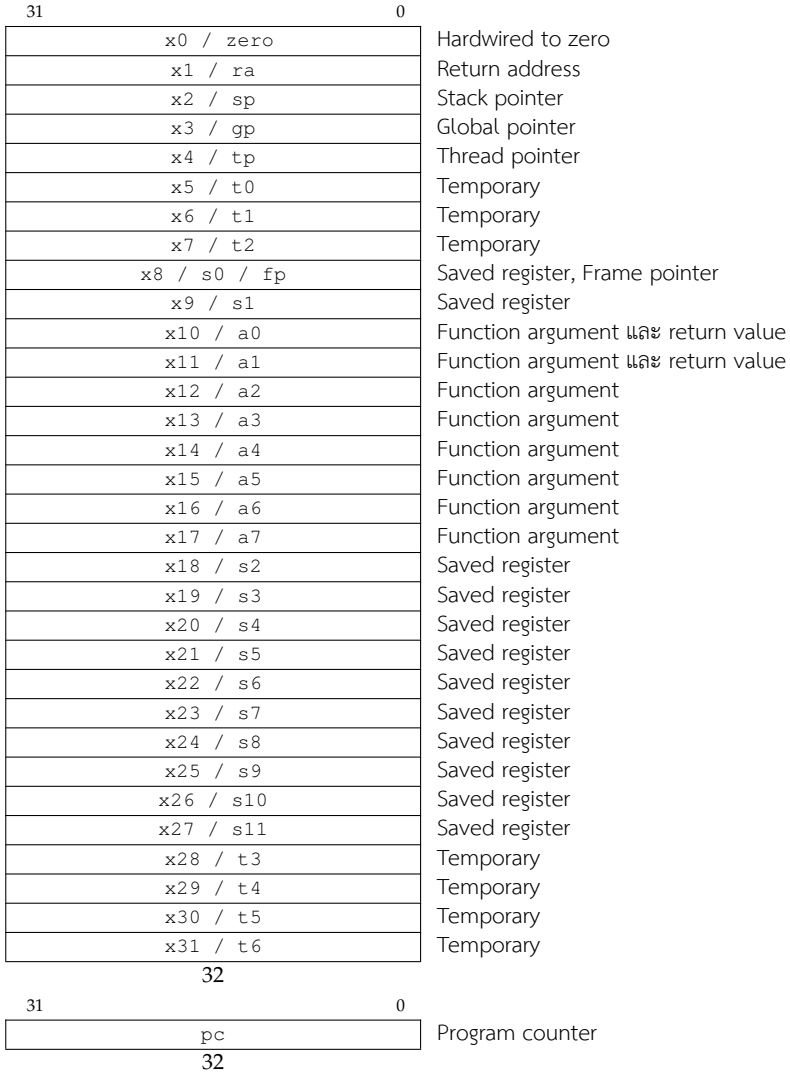
รูปที่ 2.4 แสดงรายการรีจิสเตอร์ของ RV32I และชื่อของรีจิสเตอร์ตามที่กำหนดโดยอินเตอร์เฟซไบนารีแอปพลิเคชัน (ABI) ของ RISC-V เราจะใช้ชื่อตาม ABI ในตัวอย่างโค้ดของเรา เพื่อให้อ่านง่ายขึ้น เพื่อช่วยในการทำงานงานของโปรแกรมเมอร์ภาษาแอสเซมบลีและผู้เขียนคอมไพเลอร์ RV32I มีรีจิสเตอร์ทั้งหมด 31 ตัว พร้อมด้วย x0 ซึ่งจะมีค่า 0 เสมอ ในขณะที่ ARM-32 มีรีจิสเตอร์เพียง 16 ตัว และ x86-32 มีเพียง 8 รีจิสเตอร์เท่านั้น!



ความเรียบง่าย

อะไรที่แตกต่าง? การกำหนดรีจิสเตอร์หนึ่งตัวให้มีค่าเป็นศูนย์ (x0 ใน RISC-V) เป็นปัจจัยสำคัญ ที่ทำให้สถาปัตยกรรมชุดคำสั่ง RISC-V เรียบง่ายขึ้นอย่างน่าประหลาดใจ รูปที่ 3.3 หน้าที่ 42 ของบทที่ 3 ให้ตัวอย่างมากมายของการดำเนินการที่เป็นคำสั่งพื้นฐานใน ARM-32 และ x86-32 ซึ่งไม่มีรีจิสเตอร์ศูนย์ แต่ใน RV32I สามารถสังเคราะห์ได้ง่าย ๆ โดยใช้รีจิสเตอร์ศูนย์เป็นตัวถูกดำเนินการ

ใน ARM-32 ตัวนับโปรแกรม (PC - Program Counter) เป็นหนึ่งใน 16 รีจิสเตอร์ ซึ่งหมายความว่าคำสั่งใด ๆ ที่เปลี่ยนรีจิสเตอร์อาจกลายเป็นคำสั่งกระโดด โดยเป็นผลข้างเคียง การที่ PC เป็นรีจิสเตอร์ทำให้การทำนายการกระโดดในฮาร์ดแวร์ซับซ้อนขึ้น ซึ่งความแม่นยำของการทำนายนี้สำคัญมาก สำหรับประสิทธิภาพของการประมวลผลแบบท่อ



รูปที่ 2.4: รีจิสเตอร์ของ RV32I บทที่ 3 อธิบายเกี่ยวกับ กฎการเรียกฟังก์ชันของ RISC-V (RISC-V calling convention) เหตุผลเบื้องหลังตัวชี้ต่างๆ `sp` (stack pointer), `gp` (global pointer), `tp` (thread pointer), `fp` (frame pointer), รีจิสเตอร์ที่บันทึก `s0–s11` (saved registers) และรีจิสเตอร์ชั่วคราว `t0–t6` (temporaries) (รูปที่ 2.1 และ ตาราง 20.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

การประมวลผลแบบท่อ (Pipelining) ถูกใช้โดยโปรเซสเซอร์เกือบทั้งหมดในปัจจุบัน ยกเว้นโปรเซสเซอร์ที่มีต้นทุนต่ำสุด เพื่อให้ได้ประสิทธิภาพที่ดี เช่นเดียวกับ สาย การ ประกอบ ใน โรงงาน การประมวลผลแบบท่อช่วยเพิ่มปริมาณงานโดยการดำเนินการคำสั่งหลายคำสั่งพร้อมกันแบบทับซ้อนกัน เพื่อให้สำเร็จ โปรเซสเซอร์จะทำนายผลลัพธ์ของการกระโดด ซึ่งสามารถทำได้ด้วยความแม่นยำมากกว่า 90% เมื่อทำนายผิด พวกมันจะต้องดำเนอนการคำสั่งใหม่ ไมโครโปรเซสเซอร์ยุคแรกมีไปป์ไลน์ 5 ขั้นตอน ซึ่งหมายถึงมี 5 คำสั่งที่ทับซ้อนกันในการดำเนินการ โปรเซสเซอร์ในปัจจุบันมีท่อมากกว่า 10 ขั้นตอน ARM v8 ซึ่งเป็นรุ่นต่อจาก ARM-32 ได้ยกเลิกการใช้ PC เป็นรีจิสเตอร์อเนกประสงค์ ซึ่ง เท่ากับยอมรับว่าการออกแบบเดิมเป็นข้อผิดพลาด

เนื่องจากทุกคำสั่งอาจเป็นการกระโดด แทนที่จะเป็นเพียง 10–20% ของคำสั่งที่ดำเนินการในโปรแกรมสำหรับ ISA ทั่วไป นอกจากนี้ยังหมายถึงการมีรีจิสเตอร์อเนกประสงค์น้อยลงหนึ่งตัว

2.4 การคำนวณจำนวนเต็มของ RV32I

ภาคผนวก A ให้รายละเอียดของคำสั่งทั้งหมดใน RISC-V รวมถึงรูปแบบและรหัสดำเนินการ ในส่วนนี้และส่วนที่คล้ายกันในบทต่อๆ ไป เราจะให้ภาพรวมของ ISA ที่เพียงพอสำหรับโปรแกรมเมอร์ภาษาแอสเซมบลีที่ควรรู้ รวมถึงเน้นคุณสมบัติที่แสดงถึงเมตริกของ ISA ทั้งเจ็ด จากบทที่ 1

คำสั่งเลขคณิตพื้นฐาน (`add`, `sub`), คำสั่งลอจิก (`and`, `or`, `xor`) และคำสั่งการเลื่อนบิต (`sll`, `srl`, `sra`) ในรูปที่ 2.1 เป็นไปตามที่คาดหวังใน ISA ทั่วไป คำสั่งเหล่านี้อ่านค่าขนาด 32 บิตสองค่าจากรีจิสเตอร์ และเขียนผลลัพธ์ขนาด 32 บิต ไปยังรีจิสเตอร์ปลายทาง RV32I ยังมีคำสั่งที่ใช้ค่าแบบทันที (immediate) สำหรับคำสั่งเหล่านี้ด้วย ต่างจาก ARM-32 ค่าแบบทันทีใน RISC-V จะถูกจะขยายเครื่องหมาย (sign-extended) เสมอ เพื่อให้สามารถเป็นค่าลบได้เมื่อจำเป็น ซึ่งเป็นเหตุผลว่าทำไมถึงไม่จำเป็นต้องมีคำสั่ง ใช้ค่าแบบทันทีของ `sub`



ความเรียบง่าย

โปรแกรมสามารถสร้างค่าแบบบูลีนจากผลลัพธ์ของการเปรียบเทียบ เพื่อรองรับกรณีดังกล่าว RV32I มีคำสั่ง *set less than* ซึ่งจะกำหนดให้รีจิสเตอร์ปลายทางมีค่าเป็น 1 หากตัวถูกดำเนินการตัวแรกน้อยกว่าตัวที่สอง หรือ 0 หากไม่เป็นเช่นนั้น ตามที่คาดไว้ มีทั้งคำสั่งที่ใช้เครื่องหมาย `slt` และคำสั่งที่ไม่ใช้เครื่องหมาย (`sltu`) สำหรับจำนวนเต็มแบบมีเครื่องหมายและไม่มีเครื่องหมาย รวมถึงคำสั่งใช้ค่าแบบทันที สำหรับทั้งสองคำสั่ง (`slti`, `sltiu` ดังที่เราจะเห็นต่อไป ในขณะที่การกระโดดแบบมีเงื่อนไขใน RV32I สามารถตรวจสอบความสัมพันธ์ทั้งหมดระหว่างรีจิสเตอร์สองตัวได้ แต่บางนิพจน์เงื่อนไขอาจเกี่ยวข้องกับความสัมพันธ์ระหว่างคู่รีจิสเตอร์หลายคู่ คอมไพเลอร์หรือโปรแกรมเมอร์ภาษาแอสเซมบลีสามารถใช้คำสั่ง `slt` ร่วมกับคำสั่งลอจิก `and`, `or` และ `xor` เพื่อแก้ไขนิพจน์เงื่อนไขที่ซับซ้อนมากขึ้น



การเขียนโปรแกรม

คำสั่งการคำนวณจำนวนเต็มที่เหลืออีกสองคำสั่งใน รูปที่ 2.1 ช่วยในการแอสเซมบลีและการลิงก์ คำสั่ง (*Load upper immediate*) (`lui`) โหลดค่าคงที่ 20 บิตเข้าไปใน 20 บิตที่มีนัยสำคัญสูงสุดของรีจิสเตอร์ โดยสามารถตามด้วยคำสั่งใช้ค่าแบบทันทีมาตรฐานเพื่อสร้างค่าคงที่ขนาด 32 บิตโดยใช้เพียงสองคำสั่ง RV32I ขนาด 32 บิต คำสั่ง (*Add upper immediate to PC*) (`auipc`) รองรับลำดับคำสั่ง 2 คำสั่ง เพื่อเข้าถึงออฟเซ็ต (offset) ใด ๆ จากตัวนับโปรแกรม (PC) สำหรับทั้งการถ่ายโอนการควบคุมและการเข้าถึงข้อมูล การรวมกันของ `auipc` กับค่าคงที่ 12 บิตในคำสั่ง `jalr` (ดูด้านล่าง) สามารถถ่ายโอนการควบคุมไปยังที่อยู่ใด ๆ ที่สัมพันธ์กับ PC ขนาด 32 บิต ในขณะที่ `auipc` รวมกับออฟเซ็ตแบบทันที 12 บิตในคำสั่งโหลดหรือจัดเก็บข้อมูลสามารถเข้าถึงที่อยู่ข้อมูลใดๆ สัมพัทธ์กับ PC ขนาด 32 บิต



ความเรียบง่าย

อะไรที่แตกต่าง? ประการแรก ไม่มีคำสั่งการคำนวณจำนวนเต็มสำหรับข้อมูลขนาดไบต์ (byte) หรือครึ่งคำ (half-word) การดำเนินการทั้งหมดจะใช้ความกว้างเต็มของรีจิสเตอร์เสมอ การเข้าถึงหน่วยความจำใช้พลังงานมากกว่าการดำเนินการเลขคณิตหลายเท่า ดังนั้น การเข้าถึงข้อมูลขนาดแคบสามารถประหยัดพลังงานได้อย่างมาก แต่การดำเนินการขนาดแคบไม่ได้ช่วยในด้านนี้ ARM-32 มีคุณสมบัติที่ไม่ธรรมดา คือมีตัวเลือกให้เลื่อน (shift) หนึ่งในตัวถูกดำเนินการในคำสั่งเลขคณิต-ลอจิกส่วนใหญ่ ซึ่งทำให้เส้นทางข้อมูล (datapath) ซับซ้อนขึ้นและมักไม่จำเป็น [Hohl and Hinds 2016]; ในขณะที่ RV32I มีคำสั่งเลื่อนบิตแยกต่างหาก

ISA ต่อจาก ARM-32 คือ ARM v8 ได้ยกเลิกตัวเลือกการดำเนินการเลื่อน ออกจากคำสั่ง ALU ซึ่งบ่งบอกอีกครั้งว่ามันเป็นความผิดพลาดสำหรับ ARM-32



ค่าใช้จ่าย

นอกจากนี้ RV32I ยังไม่มีคำสั่งคูณและหาร คำสั่งเหล่านี้รวมอยู่ในส่วนขยายที่เป็นตัวเลือก RV32M (ดูบทที่ 4) ซึ่งแตกต่างจาก ARM-32 และ x86-32 ชุดซอฟต์แวร์เต็มรูปแบบของ RISC-V สามารถทำงานได้โดยไม่ต้องใช้คำสั่งเหล่านี้ ซึ่งช่วยลดขนาดของชิปในระบบฝังตัว (embedded chips) สถาปนิกของ MIPS-32 สมมติว่าการใช้งานหน่วยคูณ-หารไม่ได้ใช้การประมวลผลแบบท่อ (non-pipelined) ซึ่งอาจใช้เวลาหลายสิบรอบสัญญาณนาฬิกา การกำหนดสถาปัตยกรรมนี้ทำให้เกิดปัญหาเมื่อเริ่มการคูณใหม่ทันทีหลังจากการสูญเสียผลคูณก่อนหน้า ดังนั้น หากไม่ระวัง เมื่อเกิดการขัดจังหวะ (interrupt) โปรแกรมเมอร์อาจเห็นผลลัพธ์ที่อธิบายไม่ได้ [Sweetman 2010] แม้ว่าจะไม่ใช่ปัญหาด้านฮาร์ดแวร์ แต่*แอสเซมเบลอร์* ของ MIPS-32 อาจแทนที่การคูณด้วยลำดับของการเลื่อนบิตและการบวก เพื่อพยายามปรับปรุงประสิทธิภาพ ซึ่งอาจทำให้โปรแกรมเมอร์สับสน เมื่อเห็นคำสั่งที่ดำเนินการแต่ไม่พบในโปรแกรมภาษาแอสเซมบลี RV32I ยังละเว้นคำสั่งหมุนบิตและการตรวจจับการล้นของการคำนวณ (Arithmetic Overflow) จำนวนเต็ม ซึ่งทั้งสองอย่างนี้สามารถคำนวณได้ด้วยคำสั่ง RV32I เพียงไม่กี่คำสั่ง (ดู หัวข้อ 2.6)

■ ***รายละเอียดเพิ่มเติม: ใช้*** `xor` ***ช่วยให้เกิดกลเม็ดวิเศษ***

คุณสามารถสลับค่าของสองตัวแปรได้โดยไม่ต้องใช้รีจิสเตอร์กลาง! โค้ดนี้สลับค่าของรีจิสเตอร์ x1 และ x2 เราให้ผู้อ่านพิสูจน์ด้วยตัวเอง คำใบ้ : การดำเนินการ XOR มีคุณสมบัติ สลับที่ได้ ($a \oplus b = b \oplus a$), เชื่อมโยงได้ ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), เป็น ผกผันของตัวเอง ($a \oplus a = 0$), และมีเอกลักษณ์ ($a \oplus 0 = a$)

```
xor  x1,x1,x2 # x1' == x1^x2, x2' == x2
xor  x2,x1,x2 # x1' == x1^x2, x2' == x1'^x2 == x1^x2^x2 == x1
xor  x1,x1,x2 # x1'' == x1'^x2' == x1^x2^x1 == x1^x1^x2 == x2, x2' == x1
```

ถึงแม้ว่าจะน่าสนใจ แต่ชุดรีจิสเตอร์ที่มากมายของ RISC-V มักทำให้คอมไพเลอร์สามารถหา รีจิสเตอร์ชั่วคราวได้ ดังนั้นจึงไม่ค่อยใช้เทคนิค XOR-swap นี้

2.5 โหลดและจัดเก็บข้อมูลใน RV32I

นอกจากการให้คำสั่งโหลดและบันทึกสำหรับข้อมูลเวิร์ดขนาด 32 บิต (`lw`, `sw`) แล้ว รูปที่ 2.1 ยังแสดงให้เห็นว่า RV32I มีคำสั่งโหลดสำหรับไบต์และฮาล์ฟเวิร์ด (halfwords) ทั้งแบบมีเครื่องหมายและไม่มีเครื่องหมาย (`lb`, `lbu`, `lh`, `lhu`) รวมถึงคำสั่งเก็บข้อมูลสำหรับไบต์และฮาล์ฟเวิร์ด (`sb`, `sh`) ไบต์และฮาล์ฟเวิร์ดแบบมีเครื่องหมายจะถูกขยายเครื่องหมาย (sign-extended) เป็น 32 บิตก่อนเขียนลงในรีจิสเตอร์ปลายทาง การขยาย


ความเรียบง่าย

ข้อมูลขนาดแคบ 12 บิตนี้ ช่วยให้คำสั่งการคำนวณจำนวนเต็มที่ตามมาสามารถทำงานได้อย่างถูกต้องกับข้อมูลทั้ง 32 บิต แม้ว่าประเภทข้อมูลตามธรรมชาติจะมีขนาดแคบกว่า ไบต์และฮาล์ฟเวิร์ด (halfwords) แบบไม่มีเครื่องหมาย ซึ่งมีประโยชน์สำหรับข้อความและจำนวนเต็มแบบไม่มีเครื่องหมาย จะถูกขยายศูนย์ (zero-extended) เป็น 32 บิตก่อนเขียนลงในรีจิสเตอร์ปลายทาง

โหมดการเข้าถึงตำแหน่งที่อยู่เพียงโหมดเดียวสำหรับการโหลดและเก็บข้อมูลคือการบวกค่าแบบทันทีขนาด 12 บิตที่ขยายเครื่องหมาย (sign-extended) แล้วเข้ากับรีจิสเตอร์ ซึ่งเรียกว่า โหมดการกำหนดที่อยู่แบบ displacement ใน x86-32 [Irvine 2014]

อะไรที่แตกต่าง? RV32I ไม่มีโหมดการกำหนดที่อยู่ที่ซับซ้อนของ ARM-32 และ x86-32 นอกจากนี้ โหมดการกำหนดที่อยู่ทั้งหมดของ ARM-32 ไม่สามารถใช้ได้กับทุกประเภทข้อมูล แต่การกำหนดที่อยู่ใน RV32I ไม่เลือกปฏิบัติต่อประเภทข้อมูลใดๆ RISC-V สามารถเลียนแบบโหมดการกำหนดที่อยู่บางแบบของ x86 ได้ ตัวอย่างเช่น การตั้งค่าฟิลด์ค่าแบบทันทีเป็น 0 จะให้ผลเช่นเดียวกับโหมดการกำหนดที่อยู่แบบอ้อมผ่านรีจิสเตอร์ ต่างจาก x86-32 ที่ RISC-V ไม่มีคำสั่งสแต็กพิเศษ โดยการใช้หนึ่งใน 31 รีจิสเตอร์เป็นตัวชี้สแต็ก (stack pointer) (ดูรูปที่ 2.4) โหมดการกำหนดที่อยู่มาตรฐาน จะให้ประโยชน์ส่วนใหญ่ของคำสั่ง `push` และ `pop` ได้โดยไม่ต้องเพิ่มความซับซ้อนให้กับ ISA ต่างจาก MIPS-32 โดย RISC-V ปฏิเสธการใช้ delayed load ซึ่งคล้ายคลึงกับแนวคิดของการกระโดดแบบล่าช้า (delayed branches) โดย MIPS-32 นิยามใหม่ให้คำสั่งโหลดทำให้ข้อมูลไม่พร้อมใช้งาน จนกว่าจะผ่านไปสองคำสั่ง ซึ่งข้อมูลจะปรากฏในไปป์ไลน์ห้าขั้นตอน (five-stage pipeline) ขึ้นอยู่กับคอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลีที่จะต้องวางคำสั่ง NOP หลังคำสั่งโหลด หากไม่มีคำสั่งที่มีประโยชน์ใดๆ ที่สามารถย้ายมาใส่ในช่องหน่วงเวลา (delay slot) มันจะไม่เกิดประโยชน์ใดๆ สำหรับไปป์ไลน์ที่ยาวขึ้นซึ่งเกิดขึ้นในภายหลัง

ในขณะที่ ARM-32 และ MIPS-32 ต้องการให้ข้อมูลในหน่วยความจำถูกจัดเรียงตามขอบเขตขนาดข้อมูลตามธรรมชาติ (naturally aligned) แต่ RISC-V ไม่กำหนดข้อบังคับนี้ การเข้าถึงที่ไม่จัดเรียง (misaligned accesses) บางครั้งจำเป็นเมื่อต้องพอร์ตโค้ดเก่า (legacy code) ตัวเลือกหนึ่งคือ การไม่อนุญาตให้มีการเข้าถึงที่ไม่จัดเรียงใน ISA พื้นฐาน และจัดหาคำสั่งแยกต่างหากเพื่อรองรับการเข้าถึงที่ไม่จัดเรียง เช่น คำสั่ง Load Word Left และคำสั่ง Load Word Right ของ MIPS-32 อย่างไรก็ตาม ตัวเลือกนี้จะทำให้การเข้าถึงรีจิสเตอร์มีความซับซ้อน เนื่องจากคำสั่ง `lwl` และ `lwr` ต้องเขียนข้อมูลเพียงบางส่วนของรีจิสเตอร์แทนการเขียนทั้งรีจิสเตอร์ การกำหนดให้คำสั่งโหลดและเก็บข้อมูลทั่วไปรองรับการเข้าถึงที่ไม่จัดเรียง (misaligned accesses) กลับทำให้การออกแบบโดยรวมง่ายขึ้น


การเขียนโปรแกรม


ค่าใช้จ่าย

■ ***รายละเอียดเพิ่มเติม: ลำดับไบต์ (Endianness)***

RISC-V เลือกใช้ลำดับไบต์แบบ *little-endian* เนื่องจากเป็นที่นิยมในเชิงพาณิชย์: ระบบ x86-32 ทั้งหมด รวมถึง Apple iOS, Google Android OS และ Microsoft Windows สำหรับ ARM ล้วนใช้ little-endian เนื่องจากลำดับไบต์มีความสำคัญเฉพาะเมื่อเข้าถึงข้อมูลเดียวกันทั้งในรูปแบบเวิร์ด (word) และแบบไบต์ (bytes) ดังนั้น endianness จึงส่งผลกระทบต่อโปรแกรมเมอร์เพียงเล็กน้อย

■ *รายละเอียดเพิ่มเติม: ความสอดคล้องของหน่วยความจำ*

ความสอดคล้องของหน่วยความจำในสถาปัตยกรรมกำหนดว่า เมื่อใดที่ค่าที่ถูกเขียนจะถูกส่งคืนโดยการอ่านเมื่อมีหลายเธรดทำงานพร้อมกัน แม้ว่าประเด็นนี้ยังอยู่ในการพิจารณาโดยองค์กรของ RISC-V แต่ซอฟต์แวร์ที่เขียนตามโมเดลหน่วยความจำของ ARM-64 จะสามารถทำงานได้บนโปรเซสเซอร์ RISC-V

2.6 การกระโดดแบบมีเงื่อนไขใน RV32I

RV32I สามารถเปรียบเทียบรีจิสเตอร์สองตัวและกระโดดตามผลลัพธ์ หากทั้งสองเท่ากัน (`beq`), ไม่เท่ากัน (`bne`), มากกว่าหรือเท่ากับ (`bge`) หรือน้อยกว่า (`blt`) สองกรณีหลังนี้เป็นการเปรียบเทียบแบบมีเครื่องหมาย แต่ RV32I ยังมีคำสั่งที่ใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย: `bgeu` และ `bltu` ความสัมพันธ์ที่เหลืออีกสองแบบ (มากกว่า และน้อยกว่าหรือเท่ากับ) สามารถตรวจสอบได้ง่ายๆ โดยการกลับตัวถูกดำเนินการ เนื่องจาก $x < y$ หมายถึง $y > x$ และ $x \geq y$ หมายถึง $y \leq x$

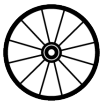
คำ สั่ง `bltu` ช่วย ให้ สามารถ ตรวจ สอบ ขอบเขต ของ อาร์เรย์ แบบมีเครื่องหมายได้ด้วย คำสั่งเดียว เนื่องจากดัชนีที่เป็นลบใดๆ จะถูกเปรียบเทียบว่ามากกว่าขอบเขตที่ไม่เป็นลบใดๆ!

เนื่องจากคำสั่งใน RISC-V ต้องมีความยาวเป็นผลคูณของสองไบต์—ดูบทที่ 7 เพื่อเรียนรู้เกี่ยวกับคำสั่งขนาด 2 ไบต์ที่เป็นตัวเลือก— โหมดการกำหนดที่อยู่สำหรับการกระโดดจะคูณค่าแบบทันทีขนาด 12 บิตด้วย 2 ขยายเครื่องหมาย (sign-extend) แล้วบวกเข้ากับตัวนับโปรแกรม (PC) การกำหนดที่อยู่สัมพัทธ์กับ PC (PC-Relative Addressing) ช่วยให้โค้ดเป็นอิสระจากตำแหน่ง (position-independent code) และลดงานของลิงเกอร์และโลดเดอร์ (ดูบทที่ 3)



การเขียนโปรแกรม

อะไรที่แตกต่าง? ดังที่กล่าวไว้ก่อนหน้านี้ RISC-V ไม่รวมการกระโดดแบบหน่วงเวลา (delayed branch) ของ MIPS-32, Oracle SPARC และอื่นๆ [ASK ANDREW: Setting aside the cost of delayed branches to the compiler and hardware, it makes the MIPS-32 code much harder to understand (see Figure 2.10)] นอกจากนี้ ยังหลีกเลี่ยงการใช้รหัสเงื่อนไขของ ARM-32 และ x86-32 สำหรับการกระโดดแบบมีเงื่อนไข รหัสเงื่อนไขเพิ่มสถานะพิเศษที่ถูกตั้งค่าโดยคำสั่งส่วนใหญ่โดยปริยาย ซึ่งทำให้การคำนวณการพึ่งพาสำหรับการประมวลผลแบบนอกลำดับ ซับซ้อนโดยไม่จำเป็น


ความเรียบง่าย

สุดท้าย RV32I ยังไม่มีคำสั่งลูปของ x86-32: `loop`, `loope`, `loopz`, `loopne`, `loopnz`

■ *รายละเอียดเพิ่มเติม: การบวกแบบมัลติเวิร์ดโดยไม่ใช้รหัสเงื่อนไข*

การบวกนี้ใน RV32I ทำได้โดยใช้คำสั่ง `sltu` เพื่อคำนวณหาค่าแคร์รี่ (carry-out) ดังนี้:

```
add  a0,a2,a4 # บวก 32 บิตล่าง: a0 = a2 + a4
sltu a2,a0,a2 # a2' = 1 ถ้า (a2+a4) < a2, a2' = 0 หากไม่เป็นเช่นนั้น
add  a5,a3,a5 # บวก 32 บิตบน: a5 = a3 + a5
add  a1,a2,a5 # บวกค่าแคร์รี่จาก 32 บิตล่าง
```

■ *รายละเอียดเพิ่มเติม: การอ่าน PC*

สามารถรับค่า PC ปัจจุบันได้โดยตั้งค่าฟิลด์ค่าแบบทันทีของ `auipc` (ฟิลด์ U-immediate) เป็น 0 ใน x86-32 การอ่าน PC ต้องเรียกฟังก์ชัน (ซึ่งจะผลัก PC ลงในสแต็ก) จากนั้นฟังก์ชันที่ถูกเรียกจะอ่าน PC ที่ถูกผลักจากสแต็ก และสุดท้ายคืนค่า PC (โดยการดึงออกจากสแต็ก) ดังนั้น การอ่าน PC ปัจจุบันใน x86-32 ต้องใช้ 1 การเก็บ (store), 2 การโหลด (loads), และ 2 การกระโดดที่ดำเนินการ (taken jumps)!

■ *รายละเอียดเพิ่มเติม: การตรวจสอบการล้นด้วยซอฟต์แวร์*

โปรแกรมส่วนใหญ่ (แต่ไม่ทั้งหมด) มองข้ามการล้นของการคำนวณจำนวนเต็ม ดังนั้น RISC-V จึงพึ่งพาการตรวจสอบการล้นด้วยซอฟต์แวร์ สำหรับการบวกแบบไม่มีเครื่องหมาย (unsigned addition) ต้องใช้เพียงคำสั่งกระโดดเพิ่มเติมหนึ่งคำสั่งหลังการบวก: `addu t0, t1, t2; bltu t0, t1, overflow`
สำหรับการบวกแบบมีเครื่องหมาย (signed addition) หากรู้เครื่องหมายของตัวถูกดำเนินการตัวหนึ่ง การตรวจสอบการล้นต้องใช้เพียงคำสั่งกระโดดหนึ่งคำสั่งหลังการบวก:
`addi t0, t1, +imm; blt t0, t1, overflow`
กรณีนี้ครอบคลุมการบวกทั่วไปที่ใช้ตัวถูกดำเนินการแบบทันที (immediate operand)
สำหรับการบวกแบบมีเครื่องหมายทั่วไป ต้องใช้คำสั่งเพิ่มเติมสามคำสั่งหลังการบวก โดยพิจารณาว่าผลบวกควรจะน้อยกว่าหนึ่งในตัวถูกดำเนินการก็ต่อเมื่อตัวถูกดำเนินการอีกตัวเป็นลบ:

```
add t0, t1, t2  # t0 = t1 + t2
slti t3, t2, 0  # t3 = (t2<0) : t3 = 1 ถ้า t2<0, t3=0 หากไม่เป็นเช่นนั้น
slt t4, t0, t1  # t4 = (t1+t2<t1) : t4=1 ถ้า t0<t1, t4=0 หากไม่เป็นเช่นนั้น
bne t3, t4, overflow # กระโดดไปที่ overflow ถ้า (t2<0) && (t1+t2>=t1)
                     #                    || (t2>=0) && (t1+t2<t1)
```

2.7 การกระโดดแบบไม่มีเงื่อนไขใน RV32I



ความเรียบง่าย

คำสั่ง *jump and link* (`jal`) ในรูปที่ 2.1 มีหน้าที่สองประการ เพื่อรองรับการเรียกฟังก์ชัน (procedure calls) คำสั่งนี้จะบันทึกที่อยู่ของคำสั่งถัดไป PC+4 ลงในรีจิสเตอร์ปลายทาง ซึ่งโดยปกติคือรีจิสเตอร์ที่อยู่คืนค่า `ra` (ดูรูปที่ 2.4) เพื่อรองรับการกระโดดแบบไม่มีเงื่อนไข เราใช้ รีจิสเตอร์ศูนย์ (x0) แทน `ra` เป็นรีจิสเตอร์ปลายทาง เนื่องจาก x0 ไม่สามารถเปลี่ยนค่าได้ เช่นเดียวกับการกระโดดแบบมีเงื่อนไข (branches) คำสั่ง `jal` จะคูณตำแหน่งที่อยู่การกระโดดขนาด 20 บิตด้วย 2 ขยายเครื่องหมาย (sign-extend) แล้วบวกผลลัพธ์เข้ากับ PC เพื่อสร้างที่อยู่สำหรับการกระโดด

คำสั่งที่ใช้รีจิสเตอร์ของคำสั่งกระโดดและเชื่อมโยง (`jalr`) ก็มีจุดประสงค์หลากหลายเช่นกัน สามารถใช้เรียกฟังก์ชันไปยังที่อยู่ที่คำนวณแบบไดนามิก (dynamically calculated address) หรือใช้สำหรับการคืนค่าจากฟังก์ชัน (procedure return) โดยเลือก `ra` เป็นรีจิสเตอร์ต้นทาง และใช้รีจิสเตอร์ศูนย์ (x0) เป็นรีจิสเตอร์ปลายทางอีกครั้ง คำสั่ง Switch หรือ case ซึ่งคำนวณตำแหน่งที่อยู่สำหรับการกระโดด ก็สามารถใช้คำสั่ง `jalr` โดยกำหนดรีจิสเตอร์ศูนย์เป็นรีจิสเตอร์ปลายทางได้เช่นกัน

อะไรที่แตกต่าง? RV32I หลีกเลี่ยงคำสั่งเรียกฟังก์ชันที่ซับซ้อน เช่น คำสั่ง `enter` และ `leave` ของ x86-32 หรือ *หน้าต่างรีจิสเตอร์* (register windows) ที่พบใน Intel Itanium, Oracle SPARC และ Cadence Tensilica

หน้าต่างรีจิสเตอร์ ช่วยเร่งการเรียกฟังก์ชันโดยมีรีจิสเตอร์มากกว่า 32 ตัว ฟังก์ชันใหม่จะได้รับชุดหรือหน้าต่าง ของรีจิสเตอร์ 32 ตัวเมื่อมีการเรียกฟังก์ชันเพื่อ ส่งผ่าน อาร์กิวเมนต์ หน้าต่างเหล่านี้จะทับซ้อนกัน ซึ่งหมายความว่าบางรีจิสเตอร์จะอยู่ในหน้าต่างที่อ่

```
void insertion_sort(long a[], size_t n)
{
  for (size_t i = 1, j; i < n; i++) {
    long x = a[i];
    for (j = i; j > 0 && a[j-1] > x; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

รูปที่ 2.5: การเรียงลำดับแบบแทรกในภาษา C แม้ว่าว่าจะเรียบง่าย แต่การเรียงลำดับแบบแทรก (Insertion Sort) มีข้อดีหลายประการเมื่อเทียบกับอัลกอริทึมการเรียงลำดับที่ซับซ้อน: มันประหยัดหน่วยความจำ มีความเร็วสำหรับชุดข้อมูลขนาดเล็ก ขณะมีความสามารถในการปรับตัว (adaptive), คงที่ (stable) และทำงานแบบออนไลน์ คอมไพเลอร์ GCC สร้างโค้ดสำหรับรูปทั้งสี่ต่อไปนี้ เราได้ตั้งค่าแฟล็กการปรับแต่ง (optimization flags) เพื่อลดขนาดโค้ด เนื่องจากวิธีนี้ทำให้ได้โค้ดที่เข้าใจง่ายที่สุด

2.8 ส่วนเบ็ดเตล็ดของ RV32I

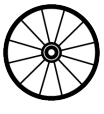
คำสั่งรีจิสเตอร์ควบคุมสถานะ (`csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi`) ในรูปที่ 2.1 ช่วยให้เข้าใจรีจิสเตอร์ที่ใช้ในการวัดประสิทธิภาพของโปรแกรมได้ง่าย ตัวนับขนาด 64 บิตเหล่านี้ ซึ่งสามารถอ่านได้ครั้งละ 32 บิต วัดเวลาแบบนาฬิกาจริง, รอบนาฬิกาที่ดำเนินการ และจำนวนคำสั่งที่เสร็จสิ้น

คำสั่ง `ecall` ใช้สำหรับร้องขอไปยังสภาพแวดล้อมการทำงานที่สนับสนุน เช่น การเรียกระบบ (system calls) โปรแกรมดีบักเกอร์ใช้คำสั่ง `ebreak` เพื่อถ่ายโอนการควบคุมไปยังสภาพแวดล้อมการดีบัก

คำสั่ง `fence` จัดลำดับการเข้าถึงอุปกรณ์ I/O และหน่วยความจำตามที่มองเห็นโดยเธรดอื่น ๆ และโดยอุปกรณ์ภายนอกหรือโคโปรเซสเซอร์ คำสั่ง `fence.i` (instruction stream) และสตรีมข้อมูล (data stream) RISC-V ไม่รับประกันว่า การเก็บข้อมูลไปยังหน่วยความจำคำสั่ง (instruction memory) จะมองเห็นได้จากการดึงคำสั่ง (instruction fetches) ในโปรเซสเซอร์เดียวกันจนกว่าคำสั่ง `fence.i` จะถูกดำเนินการ

บทที่ 11 ครอบคลุมคำสั่งระบบของ RISC-V

อะไรที่แตกต่าง? RISC-V ใช้ memory-mapped I/O แทนคำสั่ง `in`, `ins`, `insb`, `insw` และ `out`, `outs`, `outsb`, และ `outsw` ของ x86-32 การจัดการสตริงใน RISC-V ใช้การโหลดและเก็บข้อมูลแบบไบต์ แทนคำสั่งสตริงพิเศษ 16 คำสั่งของ x86-32 เช่น `rep`, `movs`, `coms`, `scas`, `lods`, ...



ความเรียบง่าย

■ ***รายละเอียดเพิ่มเติม: ส่วนขยายมาตรฐานสำหรับระบบฝังตัว RV32E***

เพื่อลดต้นทุนของคอร์ที่มีต้นทุนต่ำ RV32E เป็นชุดคำสั่งพื้นฐาน ที่มีรีจิสเตอร์น้อยลง 16 ตัว RV32E เป็นเหตุผลที่รีจิสเตอร์ที่เก็บรักษา และรีจิสเตอร์ชั่วคราว ถูกแบ่งระหว่างรีจิสเตอร์ 0-15 และ 16-31 (รูปที่ 3.2)

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC
Instructions	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

รูปที่ 2.6: จำนวนคำสั่งและขนาดโค้ดสำหรับการเรียงลำดับแบบแทรก สำหรับแต่ละ ISA สำหรับชุดคำสั่งเหล่านี้ บทที่ 7 อธิบาย ARM Thumb-2, microMIPS และ RV32C

2.9 เปรียบเทียบ RV32I, ARM-32, MIPS-32, และ x86-32

คำสั่งพื้นฐานของ RISC-V และแสดงความคิดเห็นเกี่ยวกับการเลือกออกแบบเมื่อเปรียบเทียบกับ ARM-32, MIPS-32 และ x86-32 ตอนนี้เราจะทำการเปรียบเทียบแบบตัวต่อตัว รูปที่ 2.5 แสดงการเรียงลำดับแบบแทรกในภาษา C ซึ่งจะเป็นเกณฑ์มาตรฐานของเรา รูปที่ 2.6 เป็นตารางที่สรุปจำนวนคำสั่งและจำนวนไบต์ในโค้ดการเรียงลำดับแบบแทรกสำหรับแต่ละ ISA รูปที่ 2.8 ถึง 2.11 แสดงโค้ดที่คอมไพล์สำหรับ RV32I, ARM-32, MIPS-32 และ x86-32 แม้จะเน้นความเรียบง่าย แต่รุ่นของ RISC-V ใช้จำนวนคำสั่งเท่ากันหรือน้อยกว่า และขนาดโค้ดของแต่ละสถาปัตยกรรมใกล้เคียงกันมาก ในตัวอย่างนี้ คำสั่งเปรียบเทียบและกระโดด (compare-and-execute branches) ของ RISC-V สามารถประหยัดคำสั่งได้มากเท่ากับโหมดการกำหนดที่อยู่ที่ซับซ้อนกว่าและคำสั่ง push และ pop ของ ARM-32 และ x86-32 ในรูปที่ 2.9 และ 2.11

เราได้ย้ายตัวอย่างโค้ดไปไว้หลังส่วนท้ายของเนื้อหาบท เพื่อรักษาความต่อเนื่องของการเขียนในบทนี้และบทต่อๆ ไป

2.10 ข้อสังเกตสรุป

ผู้ที่ไม่สามารถจดจำอดีตได้ย่อมถูกลิขิตให้ต้องทำซ้ำ

—จอร์จ ซานตายานา ค.ศ. 1905

รูปที่ 2.7 ใช้ตัววัดประสิทธิภาพการออกแบบ ISA เจ็ดประการ จากบทที่ 1 เพื่อจัดระเบียบบทเรียนจาก ISA ในอดีตที่ระบุไว้ในส่วนก่อนหน้า และแสดงผลลัพธ์เชิงบวกสำหรับ RV32I เรา*ไม่ได้*บอกว่า RISC-V เป็น ISA แรกที่มีผลลัพธ์เหล่านั้น ในความเป็นจริง RV32I สืบทอดคุณสมบัติดังต่อไปนี้จาก RISC-I ซึ่งเป็นที่มาของ RISC-V [Patterson 2017]:

ลำดับการสืบทอดของคำสั่งทั้งหมดใน RISC-V ถูกบันทึกไว้ใน [Chen and Patterson 2016]

- พื้นที่หน่วยความจำแบบไบต์ตำแหน่งที่อยู่ 32 บิต
- คำสั่งทั้งหมดเป็นแบบ 32 บิต
- มี 31 รีจิสเตอร์ กว้างทั้งหมด 32 บิต โดยรีจิสเตอร์ 0 ต่อวงจรถาวรเป็นศูนย์
- การดำเนินการทั้งหมดเป็นแบบรีจิสเตอร์กับรีจิสเตอร์ (ไม่มีการดำเนินการระหว่างรีจิสเตอร์ไปยังหน่วยความจำ)
- โหลด/ จัด เก็บ แบบ เวิร์ด (word) รวม ถึง การ โหลด/ จัด เก็บ ไบต์ และ ฮาล์ฟ เวิร์ด (halfword) แบบมีเครื่องหมาย และไม่มีเครื่องหมาย
- ตัวเลือกค่าโดยทันทีสำหรับคำสั่งเลขคณิต คำสั่งตรรกะ และคำสั่งเลื่อนบิตทั้งหมด
- ขยายเครื่องหมายค่าคงที่เสมอ

ผลกระทบของลินดี้ [Lin 2017]มีการสังเกตว่าอายุขัยในอนาคตของเทคโนโลยีหรือแนวคิดนั้นจะแปรผันตามอายุของมัน มันได้ผ่านการทดสอบของกาลเวลา ดังนั้นยิ่งมันอยู่รอดมาได้นานในอดีต มันก็ยิ่งมีแนวโน้มที่จะอยู่รอดได้นานในอนาคต หากสมมติฐานนี้เป็นจริง สถาปัตยกรรม RISC อาจเป็นแนวคิดที่ดีไปอีกนาน

- โหมดการกำหนดตำแหน่งที่อยู่ข้อมูลหนึ่งโหมด (รีจิสเตอร์+ค่าคงที่) และการกระโดดแบบมีเงื่อนไขสัมพันธ์กับ PC
- ไม่มีคำสั่งการคูณหรือการหาร
- คำสั่งโหลดแบบให้ค่าทันทีขนาด 20 บิตไปที่ส่วนบนของรีจิสเตอร์ เพื่อให้ค่าคงที่ 32 บิตใช้เพียงสองคำสั่ง



สง่างาม

RISC-V ได้รับประโยชน์จากการเริ่มต้นช้ากว่า หนึ่งในสี่ถึงหนึ่งในสามของศตวรรษ ซึ่งทำให้นักออกแบบสามารถปฏิบัติตามคำแนะนำของซานตายานา (Santayana) ในการยึดแนวคิดที่ดีแต่ไม่ทำซ้ำข้อผิดพลาดในอดีต—รวมถึงข้อบกพร่องของ RISC-I—ใน ISA RISC-V ปัจจุบัน นอกจากนี้ RISC-V International จะขยาย ISA อย่างช้า ๆ ผ่านส่วนขยายที่เป็นตัวเลือก เพื่อป้องกันการเติบโตแบบค่อยเป็นค่อยไปที่มากเกินไป ซึ่งเคยเป็นปัญหากับ ISA ที่ประสบความสำเร็จในอดีต

■ ***รายละเอียดเพิ่มเติม: RV32I มีเอกลักษณ์เฉพาะตัวหรือไม่?***

ไมโครโปรเซสเซอร์ในยุคแรก มีชิปแยกต่างหากสำหรับการคำนวณเลขทศนิยม ดังนั้นคำสั่งเหล่านี้จึงเป็นตัวเลือก กฎของมัวร์ทำให้ทุกอย่างรวมอยู่ในชิปเดียว และความเป็นโมดูลใน ISA ก็ค่อย ๆ หายไป การแบ่งชุด ISA เต็มรูปแบบในโปรเซสเซอร์ที่เรียบง่าย และการใช้ซอฟต์แวร์จำลอง (emulation) เมื่อเกิดการขัดข้อง (trap) มีมานานหลายทศวรรษ เช่น ใน IBM 360 model 44 และ Digital Equipment microVAX เป็นตัวอย่าง RV32I แตกต่างตรงที่ชุดซอฟต์แวร์เต็มรูปแบบต้องการเพียงคำสั่งพื้นฐานเท่านั้น ดังนั้นโปรเซสเซอร์ RV32I ไม่จำเป็นต้องขัดข้องซ้ำ ๆ สำหรับคำสั่งที่ไม่มีใน RV32G ISA ที่ใกล้เคียงที่สุดกับ RISC-V ในแง่นี้มากที่สุดน่าจะเป็น Tensilica Xtensa ซึ่งมุ่งเป้าไปที่แอปพลิเคชันแบบฝังตัว ชุดคำสั่งพื้นฐาน 80 คำสั่งของ Xtensa ออกแบบมาให้ผู้ใช้ขยายด้วยคำสั่งที่กำหนดเองเพื่อเร่งแอปพลิเคชันของตน RV32I มี ชุดคำสั่งพื้นฐานที่เรียบง่ายกว่า มีรุ่นที่อยู่ 64 บิต และมีส่วนขยายที่มุ่งเป้าไปที่ ทั้งซูเปอร์คอมพิวเตอร์และไมโครคอนโทรลเลอร์

2.11 เรียนรู้เพิ่มเติม

Lindy effect, 2017. URL `https://en.wikipedia.org/wiki/Lindy_effect`.

T. Chen and D. A. Patterson. RISC-V genealogy. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html`.

W. Hohl and C. Hinds. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, 2016.

K. R. Irvine. *Assembly language for x86 processors*. Prentice Hall, 2014.

D. Patterson. How close is RISC-V to RISC-I?, 2017.

D. Sweetman. *See MIPS run*. Morgan Kaufmann, 2010.

	ความผิดพลาดในอดีต			บทเรียนที่ได้รับ
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
ต้นทุน	รองรับการคูณจำนวนเต็ม	รองรับการหารและการคูณจำนวนเต็ม	มีการดำเนินการ 8 บิต และ 16 บิต. มีการหารและการคูณจำนวนเต็ม	ไม่มีการดำเนินการ 8 บิต และ 16 บิต. มีการคูณและการหารจำนวนเต็มในส่วนเพิ่มเติม(RV32M)
ความเรียบง่าย	ไม่มีรีจิสเตอร์ศูนย์. การดำเนินแบบมีเงื่อนไข. โหมดการเข้าถึงข้อมูลซับซ้อน. คำสั่งแบบสแต็ก (push/pop). มีการเลื่อนบิตสำหรับคำสั่งทางคณิตศาสตร์และตรรกะ	ขยายศูนย์และเครื่องหมายให้แบบให้ค่าทันที. บางคำสั่งทางคณิตศาสตร์ทำให้เกิด overflow	ไม่มีรีจิสเตอร์ศูนย์. การเรียกใช้ฟังก์ชัน/คืนค่าซับซ้อน(เข้า-ออก). คำสั่งแบบสแต็ก (push/pop). โหมดการเข้าถึงข้อมูลซับซ้อน.คำสั่งวนลูป	รีจิสเตอร์ศูนย์ x0 มีค่า 0. ค่าคงที่มีค่าเป็นบวก. มีโหมดการเข้าถึงข้อมูล. ไม่มีการดำเนินแบบมีเงื่อนไข. การเรียกใช้ฟังก์ชัน/คืนค่า และคำสั่งแบบสแต็กไม่ซับซ้อน.ไม่มีการตรวจจับคำสั่งการเกิด overflow. แยกคำสั่งการเลื่อนบิต
ประสิทธิภาพ	รหัสในการกระโดดแบบมีเงื่อนไข. รีจิสเตอร์ต้นทาง-ปลายทางแตกต่างกันในทุกรูปแบบคำสั่ง. โหลดหลายตัว. คำนวณค่าคงที่. PC เป็นรีจิสเตอร์ทั่วไป	รีจิสเตอร์ต้นทาง-ปลายทางแตกต่างกันในทุกรูปแบบคำสั่ง	รหัสในการกระโดดแบบมีเงื่อนไข. มีรีจิสเตอร์อย่างมาก 2 ต่อคำสั่ง	คำสั่งเปรียบเทียบและคำสั่งกระโดดแบบมีเงื่อนไข(ไม่รหัสในการกระโดด). มีรีจิสเตอร์อย่างมาก 3 ต่อคำสั่ง. ไม่มีการโหลดหลายตัว. รีจิสเตอร์ต้นทาง-ปลายทางคงที่ในทุกรูปแบบคำสั่ง. ค่าคงที่ทันที. PC ไม่เป็นรีจิสเตอร์ทั่วไป
การแยกสถาปัตยกรรมกับการนำไปใช้	แสดงความยาวของไปป์ไลน์เมื่อบันทึกลงใน PC เป็นรีจิสเตอร์ทั่วไป	มีการหน่วงการกระโดดและการหน่วงโหลด. รีจิสเตอร์ HI และ LO เพื่อการคูณและหารเท่านั้น	รีจิสเจอร์ไม่ใช่จีจิสเตอร์วัตถุประสงค์ทั่วไป (AX, CX, DX, DI, SI มีการใช้งานเฉพาะ)	ไม่มีการหน่วงการกระโดดและการหน่วงโหลด. จิสเตอร์วัตถุประสงค์ทั่วไป
พื้นที่ในการขยายในอนาคต	พื้นที่ opcode ว่างที่จำกัด	พื้นที่ opcode ว่างที่จำกัด		พื้นที่ opcode ว่างที่กว้าง
ขนาดโปรแกรม	มีคำสั่งขนาด 32 บิตเท่านั้น (+Thumb-2 แยกจาก ISA)	มีคำสั่งขนาด 32 บิตเท่านั้น (+microMIPS แยกจาก ISA)	มีคำสั่งขนาดบิตไม่คงที่, เป็นตัวเลือกที่ไม่ดี	มีคำสั่งขนาด 32 บิต + 16 บิต ส่วนขยาย RV32C
ง่ายในการเขียนโปรแกรม / การแปลคำสั่ง / การลิงค์	มี 15 รีจิสเตอร์เท่านั้น. ข้อมูลตรงกันในหน่วยความจำ. โหมดการเข้าถึงข้อมูลไม่สม่ำเสมอ. ตัวนับประสิทธิภาพไม่สอดคล้องกัน	ข้อมูลตรงกันในหน่วยความจำ. โหมดการเข้าถึงข้อมูลไม่สม่ำเสมอ	มี 8 รีจิสเตอร์เท่านั้น. ไม่มีโหมดการเข้าถึงข้อมูลแบบสัมพัทธ์. Inconsistent performance counters	มี 31 รีจิสเตอร์. ข้อมูลกำหนดเพิ่ม. มีโหมดการเข้าถึงข้อมูลแบบสัมพัทธ์. โหมดการเข้าถึงข้อมูลสม่ำเสมอ. ตัวนับประสิทธิภาพที่กำหนดไว้ในการออกแบบ

รูปที่ 2.7: บทเรียนที่นักออกแบบ RISC-V เรียนรู้จากข้อผิดพลาดของ ISA ในอดีต บ่อยครั้งที่บทเรียนเป็นเพียงการหลีกเลี่ยง “การปรับปรุง” ของ ISA ในอดีต บทเรียนและข้อผิดพลาดเหล่านี้ถูกจำแนกตามเมตริกของ ISA ทั้งเจ็ด จากบทที่ 1 คุณสมบัติหลายประการที่ระบุภายใต้หมวดหมู่ของต้นทุน ความเรียบง่าย และประสิทธิภาพสามารถสลับกันได้ เนื่องจากมันเป็นเรื่องของความชอบส่วนบุคคล แต่สิ่งเหล่านี้มีความสำคัญไม่ว่าจะถูกจัดอยู่ในหมวดหมู่ใด

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

```
# RV32I (19 คำสั่ง, 76 bytes, หรือ 52 bytes ด้วย RVC)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
   0: 00450693  addi  a3,a0,4    # a3 ชี้ไปที่ a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # ถ้า i < n กระโดดไปที่ Continue Outer Loop
Exit Outer Loop:
   c: 00008067  jalr  x0,x1,0    # ออกจากฟังก์ชัน
Continue Outer Loop:
  10: 0006a803  lw    a6,0(a3)   # x = a[i]
  14: 00068613  addi  a2,a3,0    # a2 เป็นพ้อยเตอร์ของ a[j]
  18: 00070793  addi  a5,a4,0    # j = i
Inner Loop:
  1c: ffc62883  lw    a7,-4(a2)  # a7 = a[j-1]
  20: 01185a63  bge   a6,a7,34   # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
  24: 01162023  sw    a7,0(a2)   # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ffc60613  addi  a2,a2,-4   # ลด a2 ชี้ไปที่ a[j]
  30: fe0796e3  bne   a5,x0,1c   # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
  34: 00279793  slli  a5,a5,0x2  # คูณ a5 ด้วย 4
  38: 00f507b3  add   a5,a0,a5   # a5 ตำแหน่งที่อยู่ของ a[j]
  3c: 0107a023  sw    a6,0(a5)   # a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00468693  addi  a3,a3,4    # เพิ่ม a3 ชี้ไปที่ a[i]
  48: fc1ff06f  jal   x0,8       # กระโดดไปที่ Outer Loop
```

รูปที่ 2.8: โค้ด RV32I สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5. ตำแหน่งที่อยู่ในรูปเลขฐานสิบหกอยู่ทางด้านซ้าย ตามด้วยโค้ดภาษาเครื่องเป็นเลขฐานสิบหกอยู่ถัดไป จากนั้นเป็นคำสั่งภาษาแอสเซมบลีและความคิดเห็น RV32I จัดสรรรีจิสเตอร์ 2 ตัวให้ชี้ไปยัง `a[j]` และ `a[j-1]` มีรีจิสเตอร์มากมาย โดยบางตัวถูกกำหนดโดย ABI สำหรับการเรียกโพรซีเดอร์ ต่างจาก ISA อื่นๆ RV32I ข้ามการบันทึกและกู้คืนรีจิสเตอร์เหล่านี้ไปยังหน่วยความจำ แม้ว่าขนาดโค้ดจะใหญ่กว่า x86-32 แต่การใช้คำสั่ง RV32C ที่เป็นตัวเลือก (ดูบทที่ 7) จะช่วยลดช่องว่างของขนาดโค้ด สังเกตว่าคำสั่งเปรียบเทียบ และการกระโดดช่วยหลีกเลี่ยงคำสั่งเปรียบเทียบทั้งสามคำสั่งที่ ARM-32 และ x86-32 ต้องการ

```
# ARM-32 (19 คำสั่ง, 76 bytes; หรือ 18 คำสั่ง/46 bytes ด้วย Thumb-2)
# r0 ชี้ไปที่ a[0], r1 คือ n, r2 คือ j, r3 คือ i, r4 คือ x
  0: e3a03001 mov  r3, #1               # i = 1
  4: e1530001 cmp  r3, r1               # i และ n (ไม่จำเป็น?)
  8: e1a0c000 mov  ip, r0               # ip = a[0]
  c: 212fff1e bxcs lr                   # ไม่คืนตำแหน่งที่อยู่
 10: e92d4030 push {r4, r5, lr}         # บันทึก r4, r5, คืนตำแหน่งที่อยู่
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!        # x = a[i] ; เพิ่ม ip
 18: e1a02003 mov  r2, r3               # j = i
 1c: e1a0e00c mov  lr, ip               # lr = a[0] (ใช้ lr เป็น scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]        # r5 = a[j-1]
 24: e1550004 cmp  r5, r4               # เปรียบเทียบ a[j-1] vs. x
 28: da000002 ble  38                   # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
 2c: e2522001 subs r2, r2, #1           # j--
 30: e40e5004 str  r5, [lr], #-4        # a[j] = a[j-1]
 34: 1afffff9 bne  20                   # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1           # i++
 3c: e1530001 cmp  r3, r1               # i และ n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14                   # ถ้า i < n กระโดดไปที่ Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}         # กู้คืน r4, r5, และ คืนตำแหน่งที่อยู่
```

รูปที่ 2.9: โค้ด ARM-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 ตำแหน่งที่อยู่ในรูปแบบเลขฐานสิบหกอยู่ทางด้านซ้าย ตามด้วยรหัสภาษาเครื่องในรูปแบบเลขฐานสิบหกอยู่ จากนั้นเป็นคำสั่งภาษาแอสเซมบลี และความคิดเห็น ARM-32 ซึ่งมีจำนวนรีจิสเตอร์จำกัด จะบันทึกสองรีจิสเตอร์ลงในสแต็กเพื่อใช้ซ้ำในภายหลังพร้อมกับที่อยู่คืนค่า (return address) มันใช้โหมดการกำหนดที่อยู่ที่ปรับสเกล (scale) ค่า `i` และ `j` ให้เป็นที่อยู่ไบต์ เนื่องจากการกระโดดแบบมีเงื่อนไขมีโอกาสเปลี่ยน ISA ระหว่าง ARM-32 และ Thumb-2 ได้ คำสั่ง `bxcs` จะตั้งค่าบิตที่มีนัยสำคัญน้อยที่สุดของตำแหน่งที่อยู่คืนค่าให้เป็น 0 ก่อนบันทึก รหัสเงื่อนไขช่วยบันทึกคำสั่งเปรียบเทียบ 1 คำสั่ง ในการตรวจสอบ `j` หลังจากลดค่าลง แต่ยังมีการเปรียบเทียบสามครั้งการในส่วนอื่น

```
# MIPS-32 (24 คำสั่ง, 96 bytes, หรือ 56 bytes ด้วย microMIPS)
# a1 คือ n, a3 เป็นตัวชี้ไปที่ a[0], v0 คือ j, v1 คือ i, t0 คือ x
   0: 24860004 addiu a2,a0,4   # a2 เป็นตัวชี้ไปที่ a[i]
   4: 24030001 li    v1,1      # i = 1
Outer Loop:
   8: 0065102b sltu  v0,v1,a1  # กำหนดให้ i < n
   c: 14400003 bnez  v0,1c     # ถ้า i < n กระโดดไปที่ Continue Outer Loop
  10: 00c03825 move  a3,a2     # a3 เป็นตัวชี้ไปที่ a[j] (ช่องเสียบเติม)
  14: 03e00008 jr    ra        # ออกจากฟังก์ชัน
  18: 00000000 nop             # ไม่ดำเนินการ
Continue Outer Loop:
  1c: 8cc80000 lw    t0,0(a2)  # x = a[i]
  20: 00601025 move  v0,v1     # j = i
Inner Loop:
  24: 8ce9fffc lw    t1,-4(a3) # t1 = a[j-1]
  28: 00000000 nop             # ไม่ดำเนินการ
  2c: 0109502a slt   t2,t0,t1  # set a[i] < a[j-1]
  30: 11400005 beqz  t2,48     # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
  34: 00000000 nop             # ไม่ดำเนินการ
  38: 2442ffff addiu v0,v0,-1  # j--
  3c: ace90000 sw    t1,0(a3)  # a[j] = a[j-1]
  40: 1440fff8 bnez  v0,24     # ถ้า j != 0 กระโดดไปที่ Inner Loop
  44: 24e7fffc addiu a3,a3,-4  # ลด a3 ชี้ไปที่ a[j] (ช่องเสียบเติม)
Exit Inner Loop:
  48: 00021080 sll   v0,v0,0x2 #
  4c: 00821021 addu  v0,a0,v0  # v0 เป็นที่อยู่ของ a[j]
  50: ac480000 sw    t0,0(v0)  # a[j] = x
  54: 24630001 addiu v1,v1,1   # i++
  58: 1000ffeb b     8         # กระโดดไปที่ Outer Loop
  5c: 24c60004 addiu a2,a2,4   # เพิ่ม a2 ชี้ไปที่ a[i] (ช่องเสียบเติม)
```

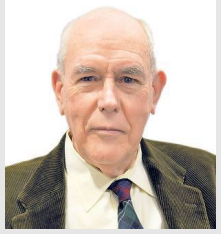
รูปที่ 2.10: โค้ด MIPS-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 ตำแหน่งที่อยู่ในรูปแบบเลขฐานสิบหกอยู่ทางด้านซ้าย ตามด้วยโค้ดภาษาเครื่องรูปแบบเลขฐานสิบหกอยู่ถัดไป จากนั้นเป็นคำสั่งภาษาแอสเซมบลี พร้อมด้วยความคิดเห็น โค้ด MIPS-32 มีคำสั่ง `nop` 3 คำสั่ง ซึ่งเพิ่มความยาวของโค้ด สองคำสั่งเกิดจาก delayed branches และหนึ่งคำสั่งเกิดจาก delayed load คอมไพเลอร์ไม่สามารถหาคำสั่งที่มีประโยชน์ มาใส่ในช่องหน่วงเวลาเหล่านั้นได้ Delayed branches ยังทำให้โค้ดเข้าใจยากขึ้น เนื่องจากคำสั่งที่ตามมาจะถูกดำเนินการเมื่อมีการกระโดดแบบมีเงื่อนไข (branch) หรือการกระโดดแบบไม่มีเงื่อนไข (jump) ตัวอย่างเช่น คำสั่งสุดท้าย (`addiu`) ที่ตำแหน่งที่อยู่ 5c เป็นส่วนหนึ่งของการวนซ้ำ แม้ว่าจะตามคำสั่งกระโดดแบบมีเงื่อนไข

```
# x86-32 (20 คำสั่ง, 45 bytes)
# eax คือ j, ecx คือ x, edx คือ i
# ตัวชี้ไปที่ a[0] ในหน่วยความจำที่ตำแหน่งที่อยู่ esp+0xc, n ในหน่วยความจำที่ตำแหน่งที่อยู่ esp+0x10
  0: 56              push esi                  # บันทึก esi บน stack (esi needed below)
  1: 53              push ebx                  # บันทึก ebx บน stack (ebx needed below)
  2: ba 01 00 00 00 mov  edx,0x1               # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]        # ecx เป็นตัวชี้ไปที่ a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]       # เปรียบเทียบ i กับ n
  f: 73 19          jae  2a <Exit Loop>       # ถ้า i >= n กระโดดไปที่ Exit Outer Loop
 11: 8b 1c 91       mov  ebx,[ecx+edx*4]      # x = a[i]
 14: 89 d0          mov  eax,edx              # j = i
Inner Loop:
 16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
 1a: 39 de          cmp  esi,ebx              # เปรียบเทียบ a[j-1] vs. x
 1c: 7e 06          jle  24 <Exit Loop>       # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
 1e: 89 34 81       mov  [ecx+eax*4],esi      # a[j] = a[j-1]
 21: 48             dec  eax                  # j--
 22: 75 f2          jne  16 <Inner Loop>      # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
 24: 89 1c 81       mov  [ecx+eax*4],ebx      # a[j] = x
 27: 42             inc  edx                  # i++
 28: eb e1          jmp  b <Outer Loop>       # กระโดดไปที่ Outer Loop
Exit Outer Loop:
 2a: 5b             pop  ebx                  # กู้คืน ค่าเดิมของ ebx จาก stack
 2b: 5e             pop  esi                  # กู้คืน ค่าเดิมของ esi จาก stack
 2c: c3             ret                       # ออกจากฟังก์ชัน
```

รูปที่ 2.11: โค้ด x86-32 สำหรับการเรียงลำดับแบบแทรกในรูปที่ 2.5 ตำแหน่งที่อยู่ในรูปแบบเลขฐานสิบหกอยู่ทางด้านซ้าย ตามด้วยรหัสภาษาเครื่องในรูปแบบเลขฐานสิบหก จากนั้นเป็นคำสั่งภาษาแอสเซมบลี ตามด้วยความคิดเห็น เนื่องจาก x86-32 มีจำนวนรีจิสเตอร์จำกัด จึงต้องบันทึกสองรีจิสเตอร์ลงในสแต็ก นอกจากนี้ ตัวแปรสองตัวที่ถูกจัดสรรให้รีจิสเตอร์ใน RV32I คือ (`n` และตัวชี้ไปที่ `a[0]`) จะถูกเก็บไว้ในหน่วยความจำแทน x86-32 ใช้โหมดการกำหนดที่อยู่แบบ Scaled Indexed ได้อย่างมีประสิทธิภาพสำหรับการเข้าถึง `a[i]` และ `a[j]` จาก 20 คำสั่ง ใน x86-32 มี 7 คำสั่งที่มีความยาวหนึ่งไบต์ ซึ่งทำให้ x86-32 มีขนาดโค้ดที่ดีสำหรับโปรแกรมง่าย ๆ นี้ มีภาษาแอสเซมบลี x86 สองรูปแบบที่นิยม: รูปแบบ Intel/Microsoft และ AT&T/Linux เราใช้รูปแบบ Intel ส่วนหนึ่งเพราะลำดับตัวถูกดำเนินการ (operand order) ตรงกับ RISC-V, ARM-32, และ MIPS-32 โดยมีรีจิสเตอร์ปลายทางอยู่ทางซ้ายและรีจิสเตอร์ต้นทางอยู่ทางขวา ในขณะที่รูปแบบ AT&T จะสลับลำดับตัวถูกดำเนินการ (และรีจิสเตอร์จะมีเครื่องหมาย % นำหน้า) เรื่องที่ดูเหมือนเล็กน้อยนี้กลายเป็นประเด็นที่เกือบจะเหมือนความเชื่อ ที่ต้องถือปฏิบัติอย่างเคร่งครัดสำหรับโปรแกรมเมอร์บางคน การตัดสินใจเลือกของเราเกิดจากเหตุผลด้านการศึกษา (pedagogy) ไม่ใช่ความเชื่อแบบดั้งเดิม (orthodoxy)

3 ภาษาแอสเซมบลี RISC-V

อีวาล ซัทเธอแลนด์ (1938–) ได้ ชื่อ ว่า เป็น บิดาแห่งกราฟิกคอมพิวเตอร์ จากการประดิษฐ์ Sketch-pad—เป็นต้นแบบของส่วนต่อประสานกราฟิกกับผู้ใช้ในคอมพิวเตอร์สมัยใหม่ในปี 1962—ซึ่งนำไปสู่การได้รับรางวัลทัวริง (Turing Award)



การแก้ปัญหาที่เราคิดว่ายากและค้นพบวิธีแก้ที่เรียบง่ายนั้นให้ความพึงพอใจอย่างมาก วิธีแก้ปัญหาที่ดีที่สุดมักจะเรียบง่ายเสมอ

—อีวาล ซัทเธอแลนด์

3.1 บทนำ

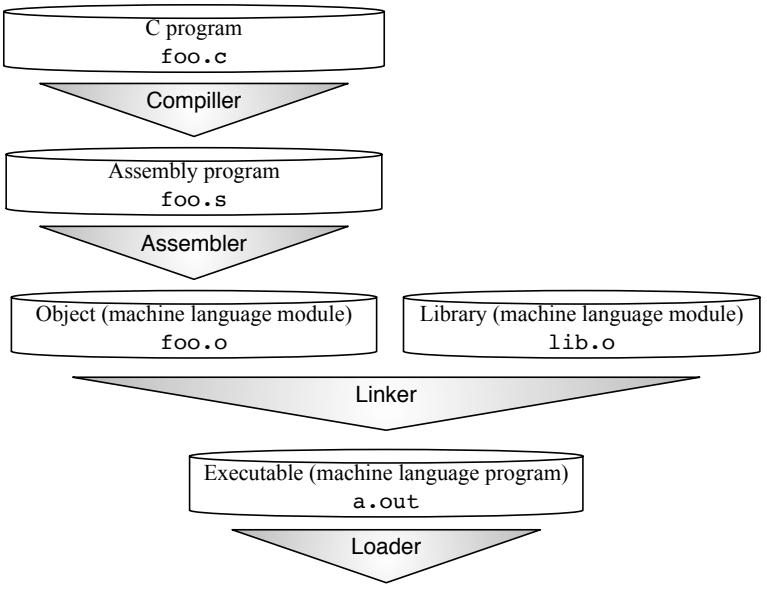
รูปที่ 3.1 แสดงขั้นตอนคลาสสิกสี่ขั้นตอนในการแปลโค้ด โดยเริ่มจากโปรแกรมภาษา C และสิ้นสุดด้วยที่โปรแกรมภาษาเครื่อง ที่พร้อมทำงานบนคอมพิวเตอร์ บทนี้ครอบคลุมขั้นตอนสามขั้นตอนสุดท้าย แต่เราเริ่มต้นด้วยบทบาทที่แอสเซมเบลอร์มีในข้อตกลงการเรียกฟังก์ชันของ RISC-V (RISC-V calling convention)

3.2 ข้อตกลงการเรียกฟังก์ชัน

การเรียกฟังก์ชัน มี 6 ขั้นตอน Patterson and Hennessy 2021

1. วางอาร์กิวเมนต์ [ในตำแหน่งหน่วยความจำหรือรีจิสเตอร์] ที่ฟังก์ชันสามารถเข้าถึงได้ [การตัดสินใจนี้เป็นส่วนหนึ่งของ API และขึ้นอยู่กับสถาปัตยกรรมโปรแกรม (RISC-V ABI)]
2. กระโดดไปที่ฟังก์ชัน (โดยใช้คำสั่ง `jal` ของ RV32I)
3. ฟังก์ชันจัดสรรพื้นที่สแต็กสำหรับตัวแปรท้องถิ่นและบันทึกรีจิสเตอร์ที่ต้องรักษาค่า
4. ฟังก์ชันดำเนินการตามที่กำหนด
5. วางผลลัพธ์ของฟังก์ชันในรีจิสเตอร์หรือในหน่วยความจำ กู้คืนรีจิสเตอร์ที่บันทึกไว้และปล่อยทรัพยากรหน่วยเก็บข้อมูล
6. เนื่องจากฟังก์ชันสามารถถูกเรียกจากหลายจุดในโปรแกรม ให้คืนการควบคุมกลับไปยังจุดเริ่มต้น (โดยใช้คำสั่ง `ret`)

เพื่อให้ได้ประสิทธิภาพที่ดี ควรพยายามเก็บตัวแปรไว้ในรีจิสเตอร์มากกว่าหน่วยความจำ แต่ในขณะเดียวกัน ควรหลีกเลี่ยงการเข้าถึงหน่วยความจำบ่อยๆ เพื่อบันทึกและคืนค่ารีจิสเตอร์เหล่านี้



รูปที่ 3.1: ขั้นตอนการแปลจากซอร์สโค้ดภาษา C ไปสู่โปรแกรมที่รันได้ เป็นขั้นตอนเชิงตรรกะ แม้ว่าบางขั้นตอนจะถูกรวมกันเพื่อเร่งกระบวนการแปล โดยใช้การตั้งชื่อส่วนขยายไฟล์ตามแบบ Unix สำหรับไฟล์แต่ละประเภท และส่วนขยายที่เทียบเท่าใน MS-DOS คือ `.C`, `.ASM`, `.OBJ`, `.LIB` และ `.EXE`

RISC-V โชคดีที่มีรีจิสเตอร์เพียงพอที่จะให้ประโยชน์ทั้งสองด้าน เก็บตัวถูกดำเนินการไว้ในรีจิสเตอร์ และลดความจำเป็นในการบันทึกและกู้คืนรีจิสเตอร์ หลักการคือ มีรีจิสเตอร์บางตัวที่ *ไม่*รับประกันว่าจะถูกรักษาค่าไว้ข้ามการเรียกฟังก์ชัน ซึ่งเรียกว่า *รีจิสเตอร์ชั่วคราว (temporary registers)* และรีจิสเตอร์บางตัวที่รับประกันว่าจะถูกรักษาค่าไว้ข้ามการเรียกฟังก์ชัน เรียกว่า *รีจิสเตอร์ที่ถูกบันทึก (saved register)* ฟังก์ชันที่ไม่เรียกฟังก์ชันอื่นเรียกว่า เรียกว่า *ฟังก์ชันใบ* เมื่อฟังก์ชันใบ มีอาร์กิวเมนต์และตัวแปรภายในเพียงไม่กี่ตัว เราสามารถเก็บทุกอย่างไว้ในรีจิสเตอร์โดยไม่ "รั่ว" ไปยังหน่วยความจำ หากเงื่อนไขเหล่านี้เป็นจริง โปรแกรมจะไม่ต้องบันทึกค่ารีจิสเตอร์ในหน่วยความจำ และที่น่าประหลาดใจคือส่วนใหญ่ของการเรียกฟังก์ชันตกอยู่ในกรณีที่ดีนี้


ประสิทธิภาพ

รีจิสเตอร์อื่นๆ ภายในการเรียกฟังก์ชัน ต้องถือว่าอยู่ในกลุ่มเดียวกับรีจิสเตอร์ที่บันทึกไว้ (saved registers) ซึ่งถูกรักษาค่าไว้ข้ามการเรียกฟังก์ชัน หรือในกลุ่มเดียวกับรีจิสเตอร์ชั่วคราว ไม่จำเป็นต้องรักษาค่าที่อยู่สำหรับการคืนค่าหรืออาร์กิวเมนต์ของฟังก์ชัน ดังนั้นรีจิสเตอร์เหล่านี้จึงเหมือนรีจิสเตอร์ชั่วคราว ผู้เรียกสามารถมั่นใจว่ารีจิสเตอร์ที่เหลือ เช่น ตัวชี้สแต็ก จะไม่เปลี่ยนแปลงข้ามการเรียกฟังก์ชัน รูปที่ 3.2 แสดงรายการชื่อของรีจิสเตอร์ตาม ABI ของ RISC-V และข้อตกลงว่ารักษาค่าไว้หรือไม่ ข้ามการเรียกฟังก์ชัน

จากโค้ดภาษา C เราสามารถเห็นโค้ดมาตรฐาน RV32I สำหรับการเข้าและออกจากฟังก์ชัน ส่วน (*prologue*) ของฟังก์ชันมีลักษณะดังนี้:

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

รูปที่ 3.2: รหัสคำสั่งแอสเซมเบลอร์สำหรับรีจิสเตอร์จำนวนเต็มและรีจิสเตอร์ทศนิยมของ RISC-V RISC-V มีรีจิสเตอร์เพียงพอที่ ABI สามารถจัดสรรรีจิสเตอร์ให้โปรซิเดอร์หรือเมธอดใช้ได้โดยไม่ต้องบันทึกหรือคืนค่า เมื่อโปรซีเดอร์หรือเมธอดเหล่านั้นไม่เรียกโปรซิเดอร์หรือเมธอดอื่น รีจิสเตอร์ที่รักษาค่าไว้ข้ามการเรียกโปรซิเดอร์เรียกว่า *callee save* ส่วนรีจิสเตอร์ที่ไม่รักษาค่าไว้เรียกว่า *caller Saved* บทที่ 5 อธิบายเกี่ยวกับรีจิสเตอร์ทศนิยม f (ตารางที่ 20.1 ของ [Waterman and Asanović 2017] เป็นที่มาของภาพนี้)

```
entry_label:
  addi sp,sp,-framesize     # จัดสรรพื้นที่สำหรับเฟรมสแต็ก
                            # โดยปรับตัวชี้สแต็ก (รีจิสเตอร์ sp)
  sw   ra,framesize-4(sp)   # บันทึกที่อยู่สำหรับการกลับ (รีจิสเตอร์ ra)
  # บันทึกรีจิสเตอร์อื่นๆ ลงในสแต็กหากจำเป็น
  … # เนื้อหาของฟังก์ชัน
```

หากมีอาร์กิวเมนต์และตัวแปรของฟังก์ชันมากเกินกว่าที่รีจิสเตอร์จะรองรับได้ *ส่วน* (*prologue*) จะจัดสรรพื้นที่บนสแต็กสำหรับ *เฟรม* ของฟังก์ชัน หลังจากงานของฟังก์ชันเสร็จสิ้น *ส่วน* (*epilogue*) จะยกเลิกสแต็กเฟรมและส่งคืนการควบคุมกลับไปที่จุดเริ่มต้น:

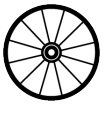
```
# คืนค่ารีจิสเตอร์จากสแต็กหากจำเป็น
lw   ra,framesize-4(sp)  # คืนค่ารีจิสเตอร์ที่อยู่สำหรับการคืนค่า (รีจิสเตอร์ ra)
addi sp,sp, framesize    # ยกเลิกการจัดสรรพื้นที่สำหรับสแต็กเฟรม
ret                      # กลับไปที่จุดที่เรียกฟังก์ชัน
```

เราจะเห็นตัวอย่างที่ปฏิบัติตาม ABI จากนี้ แต่ก่อนอื่นเราต้องอธิบายงานแอสเซมบลีที่เหลือ นอกเหนือจากการแปลงชื่อรีจิสเตอร์ของ ABI เป็นหมายเลขรีจิสเตอร์

■ ***รายละเอียดเพิ่มเติม: รีจิสเตอร์ที่บันทึกไว้และรีจิสเตอร์ชั่วคราวไม่ได้เรียงต่อกัน***

เพื่อรองรับ RV32E ซึ่งเป็นเวอร์ชันระบบฝังตัวของ RISC-V ที่มีเพียง 16 รีจิสเตอร์ (ดูบทที่ 12) โดยใช้หมายเลขรีจิสเตอร์ตั้งแต่ `x0` ถึง `x15` ดังนั้น รีจิสเตอร์ที่บันทึกไว้และรีจิสเตอร์ชั่วคราวบางตัวอยู่ในช่วงนี้ และที่เหลืออยู่ใน 16 รีจิสเตอร์สุดท้าย RV32E มีขนาดเล็กกว่า แต่ยังไม่มีคอมไพเลอร์รองรับ เนื่องจากมันไม่สอดคล้องกับ RV32I

3.3 แอสเซมบลี



ความเรียบง่าย

อินพุตสำหรับขั้นตอนนี้ใน Unix คือไฟล์ที่มีส่วนขยาย `.s` เช่น `foo.s` ส่วนใน MS-DOS คือ `.ASM`

งานของขั้นตอนแอสเซมเบลอร์ในรูปที่ 3.1 ไม่ใช่แค่การสร้างโค้ดอ็อบเจกต์จากคำสั่งที่โปรเซสเซอร์เข้าใจเท่านั้น แต่ยังขยายคำสั่งเหล่านั้นให้ครอบคลุมการดำเนินการที่เป็นประโยชน์สำหรับโปรแกรมเมอร์ภาษาแอสเซมบลีหรือผู้เขียนคอมไพเลอร์ประเภทนี้ ซึ่งอาศัยการกำหนดค่าอย่างชาญฉลาดของคำสั่งปกติ เรียกว่า *คำสั่งเทียม* (`pseudoinstructions`) รูปที่ 3.3 และ 3.4 แสดงรายการคำสั่งเทียมของ RISC-V โดยรายการในรูปแรกทั้งหมดใช้รีจิสเตอร์ x0 ที่มีค่าเป็นศูนย์เสมอ ส่วนรายการที่สองไม่ขึ้นอยู่รีจิสเตอร์ x0 ตัวอย่างเช่น คำสั่ง `ret` ที่กล่าวถึงข้างต้นเป็นคำสั่งเทียมที่แอสเซมเบลอร์แทนที่ด้วยคำสั่ง `jalr x0, x1, 0` (ดูรูปที่ 3.3) คำสั่งเทียมของ RISC-V ส่วนใหญ่ขึ้นอยู่กับ x0 จะเห็นได้ว่าการกำหนดให้หนึ่งใน 32 รีจิสเตอร์มีค่าเป็นศูนย์อย่างถาวรช่วยลดความซับซ้อนของชุดคำสั่ง RISC-V ได้อย่างมาก โดยทำให้การดำเนินการยอดนิยมหลายอย่าง—เช่น การกระโดด (jump), การคืนค่า (return), การกระโดด (jump), การกลับ (return) และการกระโดดเมื่อเท่ากับศูนย์ (branch on equal to zero)—เป็นคำสั่งเทียม

รูปที่ 3.5 แสดงโปรแกรมคลาสสิก “Hello world” ในภาษาซี คอมไพเลอร์สร้างเอาต์พุตเป็นภาษาแอสเซมบลี ในรูปที่ 3.6 โดยใช้ข้อตกลงการเรียกฟังก์ชัน ในรูปที่ 3.2

คำสั่งหลอก	คำสั่งพื้นฐาน	คำสั่งพื้นฐาน
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

รูปที่ 3.3: คำสั่งเทียม 32 คำสั่งของ RISC-V ที่ใช้รีจิสเตอร์ x0 ซึ่งเป็นรีจิสเตอร์ศูนย์ ภาคผนวก A รวมถึงคำสั่งเทียมของ RISC-V และคำสั่งจริง คำสั่งที่อ่านตัวนับ 64 บิต สามารถอ่าน 32 บิตบนใน RV32I โดยใช้เวอร์ชัน “h” ของคำสั่ง และ 32 บิตล่าง โดยใช้เวอร์ชันธรรมดา (ตารางที่ 20.2 และ 20.3 ของ [Waterman and Asanović 2017] อธิบายรูปนี้)

คำสั่งหลอก	คำสั่งพื้นฐาน	คำสั่งพื้นฐาน
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	โหลดที่อยู่ภายใน (Load local address)
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`li rd, immediate`	*Myriad sequences*	โหลดค่าคงที่ (Load immediate)
`mv rd, rs`	`addi rd, rs, 0`	คัดลอกรีจิสเตอร์ (Copy register)
`not rd, rs`	`xori rd, rs, -1`	ส่วนกลับของบิต (One's complement)
`sext.w rd, rs`	`addiw rd, rs, 0`	ส่วนกลับของบิต (One's complement)
`seqz rd, rs`	`sltiu rd, rs, 1`	ขยายเครื่องหมายของ word (Sign extend word)
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	ตั้งค่าเป็น 1 ถ้าเท่ากับศูนย์ (Set if = zero)
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	ค่าสัมบูรณ์ทศนิยมความแม่นยำเดี่ยว (Single-precision absolute value)
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	เปลี่ยนเครื่องหมายทศนิยมความแม่นยำเดี่ยว (Single-precision negate)
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	คัดลอกรีจิสเตอร์ทศนิยมความแม่นยำคู่ (Copy double-precision register)
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	ค่าสัมบูรณ์ทศนิยมความแม่นยำคู่ (Double-precision absolute value)
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	เปลี่ยนเครื่องหมายทศนิยมความแม่นยำคู่ (Double-precision negate)
`bgt rs, rt, offset`	`blt rt, rs, offset`	แบ่งแขนงถ้ามากกว่า (Branch if >)
`ble rs, rt, offset`	`bge rt, rs, offset`	แบ่งแขนงถ้าน้อยกว่าหรือเท่ากับ (Branch if ≤)
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	แบ่งแขนงถ้ามากกว่า (ไม่มีเครื่องหมาย) (Branch if >, unsigned)
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	แบ่งแขนงถ้าน้อยกว่าหรือเท่ากับ (ไม่มีเครื่องหมาย) (Branch if ≤, unsigned)
`jal offset`	`jal x1, offset`	กระโดดและเชื่อมโยง (Jump and link)
`jalr rs`	`jalr x1, rs, 0`	กระโดดและเชื่อมโยงด้วยรีจิสเตอร์ (Jump and link register)
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	เรียกซับรูทีนที่อยู่ไกล (Call far-away subroutine)
`fence`	`fence iorw, iorw`	รั้วสำหรับหน่วยความจำและ I/O ทั้งหมด (Fence on all memory and I/O)
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	สลับโหมดการปัดเศษทศนิยม (Swap FP rounding mode)
`fsrm rd, rs`	`csrrw rd, frm, rs`	สลับโหมดการปัดเศษทศนิยม (Swap FP rounding mode)
`fsflags rd, rs`	`csrrw rd, fflags, rs`	สลับแฟล็กข้อยกเว้นทศนิยม (Swap FP exception flags)

รูปที่ 3.4: คำสั่งเทียม 28 คำสั่ง ของ RISC-V ที่ไม่ขึ้นกับรีจิสเตอร์ x0 ,รีจิสเตอร์ศูนย์ สำหรับ `la`, GOT ย่อมาจาก Global Offset Table ซึ่งเก็บที่อยู่รันไทม์ของสัญลักษณ์ในไลบรารี ที่เชื่อมโยงแบบไดนามิก ภาคผนวก A รวมถึงคำสั่งเทียม RISC-V เช่นเดียวกับคำสั่งจริง (ตารางที่ 20.2 และ 20.3 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

```c
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

รูปที่ 3.5: โปรแกรม Hello World ภาษาซี (`hello.c`)

```
  .text                     # ตัวบ่งชี้: ป้อนรหัสเซกเมนต์
  .align 2                  # ตัวบ่งชี้: กด 2^2 รหัสการจัดการตำแหน่งไบต์
  .globl main               # ตัวบ่งชี้: ประกาศสัญลักษณ์หลักทั่วไป
main:                       # ตำแหน่งเริ่มฟังก์ชั่นหลัก
  addi sp,sp,-16            # จัดสรรสแต็กเฟรม
  sw ra,12(sp)              # บันทึกค่าที่อยู่
  lui a0, %hi(string1)      # คำนวณที่อยู่ของ
  addi a0,a0,%lo(string1)   #   string1
  lui a1,%hi(string2)       # คำนวณที่อยู่ของ
  addi a1,a1,%lo(string2)   #   string2
  call printf               # เรียกฟังก์ชั่นพิมพ์ call function printf
  lw ra,12(sp)              # คืนค่าตำแหน่งที่อยู่ restore return address
  addi sp,sp,16             # จัดสรรรสแตกเฟรม deallocate stack frame
  li a0,0                   # คืนค่า 0 load return value 0
  ret                       # คืนค่า return
  .section .rodata          # ตัวบ่งชี้: การป้อนเซ็กเมนต์ข้อมูลแบบอ่านอย่างเดียว
  .balign 4                 # ตัวบ่งชี้: กด 4 ข้อมูลการจัดการตำแหน่งไบต์
string1:                    # ตำแหน่งสำหรับตัวอักขระแรก label for first string
  .string "Hello, %s!\n"    # ตัวบ่งชี้: สตริงที่สิ้นสุดด้วยค่า null
string2:                    # ตำแหน่งสำหรับตัวอักขระตัวที่สอง label for second string
  .string "world"           # ตัวบ่งชี้: สตริงที่สิ้นสุดด้วยค่า null
```

รูปที่ 3.6: โปรแกรม Hello World ภาษาแอสเซมบลีใน RISC-V (`hello.s`)

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

รูปที่ 3.7: โปรแกรม Hello World ภาษาเครื่องใน RISC-V (`hello.o`) คำสั่งหกคำสั่ง ที่ถูกรันทดสอบโดยลิงเกอร์ (ตำแหน่ง 8 ถึง 1c) มีค่าเป็นศูนย์ในฟิลด์ตำแหน่งที่อยู่ของมัน ตารางสัญลักษณ์ที่รวมอยู่ในไฟล์อ็อบเจกต์บันทึกป้ายกำกับ (labels) และตำแหน่งที่อยู่ของคำสั่งทั้งหมดที่ต้องถูกแก้ไขโดยลิงเกอร์

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

รูปที่ 3.8: โปรแกรม Hello World ภาษาเครื่องใน RISC-V หลังการเชื่อมโยง ในระบบ Unix ไฟล์จะมีชื่อว่า `a.out`

และคำสั่งเทียม จากรูปที่ 3.3 และ 3.4

คำสั่งที่ขึ้นต้นด้วยจุด (.) คือ *คำสั่งแอสเซมเบลอร์* เป็นคำสั่งที่ส่งไปยังแอสเซมเบลอร์โดยตรง แทนที่จะเป็นโค้ดที่ต้องแปลโดยแอสเซมเบลอร์ คำสั่งเหล่านี้จะบอกแอสเซมเบลอร์ว่า จะวางโค้ดและข้อมูลไว้ที่ใด กำหนดค่าคงที่สำหรับข้อความ และข้อมูลใช้ในโปรแกรม และอื่นๆ ดังนี้:

- `.text`—เข้าสู่ส่วนของโค้ด
- `.align 2`—จัดตำแหน่งโค้ดต่อไปนี้ให้อยู่ในแนว 2^2 ไบต์
- `.globl main`—ประกาศสัญลักษณ์ทั่วไป ชื่อ “main”
- `.section .rodata`—เข้าสู่ส่วนข้อมูลแบบอ่านอย่างเดียว
- `.balign 4`—จัดตำแหน่งส่วนข้อมูลให้อยู่ในแนว 4 ไบต์
- `.string "Hello, %s!\n"`—สร้างสตริงที่ลงท้ายด้วยค่า null นี้
- `.string "world"`—สร้างสตริงที่ลงท้ายด้วยค่า null นี้

โปรแกรม “Hello world” มักจะเป็นโปรแกรมแรกที่ทำงานบนโปรเซสเซอร์ที่ออกแบบใหม่ สถาปนิกมักจะพิจารณาว่าใช้ระบบปฏิบัติการได้ดีพอที่จะพิมพ์คำว่า “สวัสดีชาวโลก” เพื่อเป็นสัญญาณที่ชัดเจนว่า ชิปตัวใหม่ของพวกเขาใช้งานได้ดีแล้ว พวกเขาส่งอีเมลไปยังผู้บริหารและเพื่อนร่วมงาน จากนั้นพวกเขาก็เฉลิมฉลอง

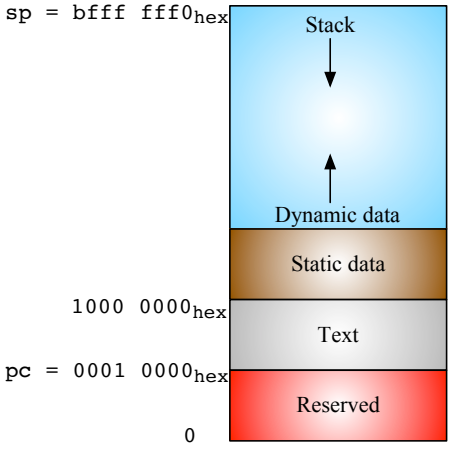
แอสเซมเบลอร์สร้างไฟล์อ็อบเจ็กต์ แสดงในรูปที่ 3.7 โดยใช้รูปแบบมาตรฐาน ELF (Executable and Linkable Format) [TIS Committee 1995]

3.4 ตัวเชื่อมโยง (Linker)

แทนที่จะคอมไพล์โค้ดทั้งหมดใหม่ทุกครั้งที่มีการเปลี่ยนแปลงไฟล์หนึ่งไฟล์ ตัวเชื่อมโยงอนุญาตให้คอมไพล์และแอสเซมเบิลไฟล์แต่ละไฟล์แยกกันได้ โดยจะ “เย็บ” โค้ดอ็อบเจ็กต์ใหม่เข้ากับโมดูลภาษาเครื่องที่มีอยู่ เช่น ไลบรารี ชื่อของมันมาจากหนึ่งในภารกิจที่ทำ ซึ่งคือการแก้ไขลิงก์ทั้งหมดของคำสั่งกระโดดและเชื่อมโยง (jump and link instructions) ในไฟล์อ็อบเจ็กต์ อันที่จริง “linker” ย่อมาจาก “link editor” ซึ่งเป็นชื่อในดั้งเดิมของขั้นตอนนี้ ในรูปที่ 3.1 ในระบบยูนิกซ์ อินพุตของตัวเชื่อมโยงคือไฟล์ที่มีส่วนขยายไฟล์ `.o`

คำสั่ง (Directive)	คำอธิบาย
`.text`	รายการต่อมาจะถูกเก็บไว้ในส่วนข้อความ (รหัสเครื่อง)
`.data`	รายการต่อมาจะถูกเก็บไว้ในส่วนข้อมูล (ตัวแปรทั่วไป)
`.bss`	รายการต่อมาจะถูกเก็บไว้ในส่วน bss (ตัวแปรทั่วไปที่ถูกกำหนดค่าเป็น 0)
`.section .foo`	รายการต่อมาจะถูกเก็บไว้ในส่วนที่ชื่อ `.foo`
`.align n`	จัดเรียงข้อมูลถัดไปให้ตรงกับขอบเขต 2^n ไบต์ เช่น `.align 2` จัดเรียงให้ตรงกับขอบเขต word (4 ไบต์)
`.balign n`	จัดเรียงข้อมูลถัดไปให้ตรงกับขอบเขต n ไบต์ เช่น `.balign 4` จัดเรียงให้ตรงกับขอบเขต word (4 ไบต์)
`.globl sym`	ประกาศว่าป้ายชื่อ sym เป็นสัญลักษณ์สากลและสามารถอ้างอิงจากไฟล์อื่นได้
`.string "str"`	เก็บสตริง str ในหน่วยความจำและสิ้นสุดด้วย null
`.byte b1,…, bn`	เก็บปริมาณ n ค่าขนาด 8 บิตในไบต์ต่อเนื่องของหน่วยความจำ
`.half w1,…,wn`	เก็บปริมาณ n ค่าขนาด 16 บิตใน halfword ต่อเนื่องของหน่วยความจำ
`.word w1,…,wn`	เก็บปริมาณ n ค่าขนาด 32 บิตใน word ต่อเนื่องของหน่วยความจำ
`.dword w1,…,wn`	เก็บปริมาณ n ค่าขนาด 64 บิตใน doubleword ต่อเนื่องของหน่วยความจำ
`.float f1,…, fn`	เก็บตัวเลขทศนิยมความแม่นยำเดี่ยว n ค่าใน word ต่อเนื่องของหน่วยความจำ
`.double d1,…, dn`	เก็บตัวเลขทศนิยมความแม่นยำคู่ n ค่าใน doubleword ต่อเนื่องของหน่วยความจำ
`.option rvc`	บีบอัดคำสั่งถัดไป (ดูบทที่เกี่ยวกับ 7)
`.option norvc`	ไม่บีบอัดคำสั่งถัดไป
`.option relax`	อนุญาตให้ลิงเกอร์ทำการผ่อนคลาย (relaxations) สำหรับคำสั่งถัดไป
`.option norelax`	ไม่อนุญาตให้ลิงเกอร์ทำการผ่อนคลายสำหรับคำสั่งถัดไป
`.option pic`	คำสั่งถัดไปเป็นรหัสที่ไม่ขึ้นกับตำแหน่ง (position-independent code)
`.option nopic`	คำสั่งถัดไปเป็นรหัสที่ขึ้นกับตำแหน่ง (position-dependent code)
`.option push`	บันทึกการตั้งค่าปัจจุบันของ `.option` ทั้งหมดลงในสแต็ก เพื่อให้ `.option pop` ถัดไปสามารถกู้คืนค่าได้
`.option pop`	กู้คืนการตั้งค่าของ `.option` ทั้งหมดจากสแต็กตามเวลาที่ทำ `.option push` ล่าสุด

รูปที่ 3.9: คำสั่งแอสเซมเบลอร์ทั่วไปของ RISC-V



รูปที่ 3.10: การจัดสรรหน่วยความจำของ RV32I สำหรับโปรแกรมและข้อมูล ตำแหน่งที่อยู่สูงอยู่ด้านบนของรูป และตำแหน่งที่อยู่ต่ำอยู่ด้านล่าง ในข้อตกลงซอฟต์แวร์ของ RISC-V ตัวชี้สแต็ก (`sp`) เริ่มต้นที่ bfff $fff0_{hex}$ และขยายไปทางข้อมูลแบบคงที่ (Static data) ส่วน text (โค้ดโปรแกรม) เริ่มต้นที่ 0001 0000_{hex} และรวมถึงไลบรารี่ที่เชื่อมโยงแบบคงที่ (statically-linked libraries) ข้อมูลแบบคงที่ (Static data) เริ่มต้นทันทีเหนือบริเวณ text ในตัวอย่างนี้ เราสมมติว่าที่อยู่นั้นคือ 1000 0000_{hex} ข้อมูลแบบไดนามิกจึงจัดสรรในภาษาซี โดยใช้ `malloc()` อยู่เหนือข้อมูลแบบคงที่เพียงเล็กน้อย เรียกว่า `heap` ซึ่งขยายขึ้นไปทางสแต็ก และรวมถึงไลบรารีที่เชื่อมโยงแบบไดนามิก (dynamically-linked libraries)

(เช่น `foo.o, libc.o`) และเอาต์พุตคือไฟล์ `a.out` สำหรับ MS-DOS อินพุต คือไฟล์ที่มีส่วนขยายไฟล์ `.OBJ` หรือ `.LIB` และเอาต์พุต คือไฟล์ `.EXE`

รูปที่ 3.10 แสดงที่อยู่ของเขตพื้นที่หน่วยความจำ ที่ถูกจัดสรรให้กับโค้ดและข้อมูลในโปรแกรม RISC-V ทั่วไป ตัวเชื่อมโยงต้องปรับที่อยู่ของโปรแกรมและข้อมูลของคำสั่งในไฟล์อ็อบเจกต์ทั้งหมดให้ตรงกับที่อยู่ในรูปนี้ งานของตัวเชื่อมโยงจะน้อยลงหากไฟล์อินพุตเป็น *โค้ดที่ไม่ขึ้นกับตำแหน่ง (position independent code) (PIC)* หมายถึงการกระโดดไปยังคำสั่งและการอ้างอิงข้อมูลภายในไฟล์จะถูกต้องไม่ว่าตัวโค้ดจะถูกวางไว้ที่ใด ตามที่กล่าวไว้ในบทที่ 2 การกระโดดสัมพัทธ์กับ PC (PC-relative branch) ของ RV32I ทำให้การสร้าง PIC ง่ายขึ้นมาก

นอกจากคำสั่งแล้ว ไฟล์อ็อบเจกต์แต่ละไฟล์ยังมีตารางสัญลักษณ์ (symbol table) ที่รวมถึงป้ายกำกับ (labels) ทั้งหมดในโปรแกรม ที่ต้องได้รับการกำหนดที่อยู่เป็นส่วนหนึ่งของกระบวนการเชื่อมโยง รายการนี้รวมถึงป้ายกำกับสำหรับข้อมูลและโค้ด รูปที่ 3.6 มีป้ายกำกับข้อมูลสองตัวที่ต้องกำหนด (`string1` และ `string2`) และป้ายกำกับโค้ดสองอันที่ต้องกำหนด (`main` และ `printf`) ตำแหน่งที่อยู่ 32 บิตนั้นยากที่จะใส่ในคำสั่ง 32 บิตเพียงคำสั่งเดียว ดังนั้นตัวเชื่อมโยงต้องปรับสองคำสั่งต่อป้ายกำกับในโค้ด RV32I ดัง

ที่แสดงในรูปที่ 3.6: คำสั่ง `lui` และ `addi` สำหรับที่อยู่ข้อมูล ส่วนคำสั่ง `auipc` และ `jalr` สำหรับที่อยู่โค้ด รูปที่ 3.8 เป็นไฟล์ `a.out` ที่ถูกลิงก์สุดท้ายของไฟล์อ็อบเจกต์ ในรูปที่ 3.7.

คอมไพเลอร์ RISC-V รองรับ ABI หลายแบบ ขึ้นอยู่กับว่ามีส่วนขยาย F และ D หรือไม่ สำหรับ RV32 ชื่อของ ABI ได้แก่ ilp32, ilp32f และ ilp32d. ilp32 หมายถึงประเภทข้อมูลในภาษา C ที่ int, long และ pointer มีขนาด 32 บิตทั้งหมด ส่วนขยายที่เป็นตัวเลือกระบุการส่งอาร์กิวเมนต์แบบทศนิยมใน ilp32 อาร์กิวเมนต์แบบทศนิยมจะถูกส่งผ่านในรีจิสเตอร์จำนวนเต็มใน ilp32f อาร์กิวเมนต์แบบทศนิยมความแม่นยำเดียว (single-precision) จะถูกส่งผ่านในรีจิสเตอร์ทศนิยมใน ilp32d อาร์กิวเมนต์แบบทศนิยมความแม่นยำสองเท่า (double-precision) ก็ถูกส่งผ่านในรีจิสเตอร์ทศนิยมด้วย

โดยธรรมชาติแล้ว การส่งผ่านอาร์กิวเมนต์จุดทศนิยมในรีจิสเตอร์ทศนิยม จำเป็นต้องมีส่วนขยาย ISA สำหรับทศนิยมที่สอดคล้องกัน คือ F หรือ D (ดูบทที่ 5) ดังนั้น การคอมไพล์โค้ดสำหรับ RV32I (แฟล็ก GCC '-march=rv32i') จะต้องใช้ ABI แบบ ilp32 (แฟล็ก GCC '-mabi=ilp32') ในทางกลับกัน การมีคำสั่งทศนิยมไม่ได้หมายความว่าแบบแผนการเรียกฟังก์ชันจะต้องใช้คำสั่งเหล่านั้น ตัวอย่างเช่น RV32IFD เข้ากันได้กับทั้งสาม ABI: ilp32, ilp32f และ ilp32d

ตัวเชื่อมโยง (linker) จะตรวจสอบว่า ABI ของโปรแกรม ตรงกับไลบรารีทั้งหมดที่ใช้ แม้ว่าคอมไพเลอร์จะรองรับการผสมผสานของส่วนขยาย ISA และ ABI ได้หลายแบบ แต่ระบบอาจติดตั้งไลบรารีเพียงไม่กี่ชุดเท่านั้น ดังนั้น ข้อผิดพลาดที่พบบ่อยคือการพยายามลิงก์โปรแกรมโดยไม่มีไลบรารีที่เข้ากันได้ติดตั้งไว้ ในกรณีนี้ ลิงเกอร์จะไม่แสดงข้อความวินิจฉัยที่ช่วยเหลือ มันจะพยายามลิงก์กับไลบรารีที่ไม่เข้ากัน แล้วแจ้งให้คุณทราบถึงความไม่เข้ากัน ข้อผิดพลาดนี้โดยทั่วไปเกิดขึ้นเมื่อทำการคอมไพล์ข้ามแพลตฟอร์ม *(cross compiling)* บนคอมพิวเตอร์เครื่องหนึ่งสำหรับคอมพิวเตอร์อีกเครื่องหนึ่ง

■ *รายละเอียดเพิ่มเติม: Linker relaxation*

คำสั่งกระโดดและเชื่อมโยง (jump and link instruction) มีฟิลด์ที่อยู่สัมพัทธ์กับ PC ขนาด 20 บิต ดังนั้นคำสั่งเดียวสามารถกระโดดได้ไกล ในขณะที่คอมไพเลอร์สร้างสองคำสั่งสำหรับแต่ละฟังก์ชันภายนอก บ่อยครั้งที่คำสั่งเดียวก็เพียงพอแล้ว การปรับแต่งนี้ช่วยประหยัดทั้งเวลาและพื้นที่ ดังนั้น ตัวเชื่อมโยงจะทำการตรวจสอบโค้ดหลายรอบเพื่อแทนที่สองคำสั่งด้วยคำสั่งเดียวเมื่อเป็นไปได้ เนื่องจากการตรวจสอบรอบหนึ่งอาจลดระยะห่างระหว่างการเรียก และฟังก์ชันให้อยู่ในขอบเขตที่คำสั่งเดียวสามารถจัดการได้ ตัวเชื่อมโยงจะปรับแต่งโค้ดต่อไปจนกว่าจะไม่มีการเปลี่ยนแปลงเพิ่มเติม กระบวนการนี้เรียกว่า การผ่อนคลายของตัวเชื่อมโยง *(Linker relaxation)* โดยชื่อนี้มาจากเทคนิคการผ่อนคลาย (relaxation techniques) สำหรับการแก้ระบบสมการ นอกจากการเรียกฟังก์ชันแล้ว ตัวเชื่อมโยงของ RISC-V ยังผ่อนคลายการกำหนดที่อยู่ข้อมูลโดยใช้ตัวชี้ส่วนกลาง (global pointer - gp) เมื่อข้อมูลอยู่ในช่วง $\pm$2 KiB จาก `gp` ซึ่งจะกำจัดคำสั่ง `lui` หรือ `auipc` ออกไป ในทำนองเดียวกัน มันยังผ่อนคลายการกำหนดที่อยู่หน่วยเก็บข้อมูลเฉพาะเธรด (thread-local storage) เมื่อข้อมูลอยู่ในช่วง $\pm$2 KiB จาก `tp` (thread pointer)

3.5 การเชื่อมโยงแบบคงที่เทียบกับแบบไดนามิก

ส่วนก่อนหน้าอธิบายเกี่ยวกับ *การเชื่อมโยงแบบคงที่* ซึ่งโค้ดไลบรารีที่อาจใช้งานทั้งหมด

สถาปนิก มัก จะ วัดประสิทธิภาพ ของโปรเซสเซอร์ โดยใช้การวัดประสิทธิภาพที่เชื่อมโยงแบบคงที่ แม้ว่าโปรแกรมจริงส่วนใหญ่จะใช้การเชื่อมโยงแบบไดนามิก ข้ออ้างคือผู้ใช้ที่สนใจด้านประสิทธิภาพควรใช้การลิงก์แบบคงที่ แต่นั่นเป็นเหตุผลที่ไม่ดีนักการเร่งประสิทธิภาพของโปรแกรมจริงมีความสมเหตุสมผลมากกว่าการมุ่งเน้นที่การทดสอบประสิทธิภาพ (benchmarks)

ถูกลิงก์และโหลดรวมกันก่อนการประมวลผล ไลบรารีดังกล่าวอาจมีขนาดค่อนข้างใหญ่ ดังนั้นการลิงก์ไลบรารีที่นิยมใช้เข้าไปในหลายโปรแกรมจะทำให้สิ้นเปลืองหน่วยความจำ นอกจากนี้ ไลบรารีจะถูกผูกมัดเมื่อถูกลิงก์—แม้ว่าจะมีการอัปเดตในภายหลังเพื่อแก้ไขข้อบกพร่อง—บังคับให้โค้ดที่ถูกลิงก์แบบคงที่ต้องใช้รุ่นเก่าที่มีข้อบกพร่อง

เพื่อหลีกเลี่ยงปัญหาทั้งสอง ระบบส่วนใหญ่พึ่งพาการเชื่อมโยงแบบไดนามิก ซึ่งฟังก์ชันภายนอกที่ต้องการจะถูกโหลดและลิงก์กับโปรแกรมเฉพาะเมื่อมีการเรียกครั้งแรกเท่านั้น หากไม่เคยถูกเรียก ฟังก์ชันนั้นจะไม่ถูกโหลดหรือเชื่อมโยงเลย ทุกการเรียกหลังจากครั้งแรกจะใช้ลิงก์ที่รวดเร็ว ดังนั้นค่าใช้จ่ายจากการเชื่อมโยงแบบไดนามิกจะเกิดขึ้นเพียงครั้งเดียว เมื่อโปรแกรมเริ่มทำงาน มันจะลิงก์กับรุ่นล่าสุดของฟังก์ชันไลบรารีที่ต้องการ ซึ่งเป็นวิธีที่ทำให้โปรแกรมได้รับรุ่นใหม่ล่าสุด นอกจากนี้ หากหลายโปรแกรมใช้ไลบรารีที่ลิงก์แบบไดนามิกเดียวกัน โค้ดในไลบรารีจะปรากฏในหน่วยความจำเพียงครั้งเดียว

โค้ดที่คอมไพเลอร์สร้างขึ้นจะคล้ายกับการลิงก์แบบคงที่ แทนที่จะกระโดดไปยังฟังก์ชันจริงโดยตรง มันจะกระโดดไปยังฟังก์ชันตัวแทน (stub function) ที่สั้น (มีสามคำสั่ง) ฟังก์ชันตัวแทนนี้จะโหลดที่อยู่ของฟังก์ชันจริงจากตารางในหน่วยความจำ แล้วกระโดดไปยังฟังก์ชันนั้น อย่างไรก็ตาม ในการเรียกครั้งแรก ตารางจะไม่มีที่อยู่ของฟังก์ชันจริง แต่จะมีที่อยู่ของรูทีนการเชื่อมโยงแบบไดนามิกแทน เมื่อถูกเรียก รูทีนการเชื่อมโยงแบบไดนามิก (dynamic linker) จะใช้ตารางสัญลักษณ์เพื่อค้นหาฟังก์ชันจริง คัดลอกมันลงในหน่วยความจำ และอัปเดตตารางให้ชี้ไปที่ฟังก์ชันจริง การเรียกครั้งต่อๆ ไปจะมีค่าใช้จ่ายเพียงสามคำสั่งของฟังก์ชันตัวแทนเท่านั้น

3.6 โหลดเดอร์ (Loader)

โปรแกรมอย่างเช่นในรูปที่ 3.8 เป็นไฟล์ไฟล์ปฏิบัติการ ที่ถูกเก็บอยู่ในหน่วยจัดเก็บข้อมูลของคอมพิวเตอร์ เมื่อโปรแกรมจะถูกเรียกใช้งาน หน้าที่ของตัวโหลด (loader) คือการโหลดโปรแกรมนั้นเข้าไปในหน่วยความจำ และกระโดดไปยังที่อยู่เริ่มต้น “โหลดเดอร์” ในปัจจุบันคือระบบปฏิบัติการ หรือกล่าวอีกนัยหนึ่ง การโหลด `a.out` เป็นหนึ่งในหลายๆ หน้าที่ของระบบปฏิบัติการ

การโหลดโปรแกรมที่เชื่อมโยงแบบไดนามิกมีความซับซ้อนขึ้นเล็กน้อย แทนที่จะเริ่มโปรแกรมโดยตรง ระบบปฏิบัติการจะเริ่มต้นด้วย การเรียกตัวเชื่อมโยงแบบไดนามิก ซึ่งจะเริ่มโปรแกรมที่ต้องการ จากนั้นจัดการการเรียกภายนอกครั้งแรกทั้งหมด คัดลอกฟังก์ชันลงในหน่วยความจำ และแก้ไขโปรแกรมหลังจากการเรียกแต่ละครั้ง เพื่อชี้ไปยังฟังก์ชันที่ถูกต้อง

3.7 ข้อสังเกตสรุป

ทำให้มันเรียบง่ายเข้าไว้ - Keep it simple, stupid
—เคลลี่ จอห์นสัน วิศวกรการบินผู้บัญญัติ “หลักการ KISS” (KISS Principle) 1960


ค่าใช้จ่าย


ประสิทธิภาพ


สง่างาม

แอสเซมเบลอร์ปรับปรุง ISA ของ RISC-V ที่เรียบง่ายด้วยคำสั่งเทียม (pseudoinstructions) 60 คำสั่ง ที่ทำให้โค้ด RISC-V อ่านง่ายและเขียนง่ายขึ้น โดยไม่เพิ่มต้นทุนฮาร์ดแวร์ การกำหนดรีจิสเตอร์หนึ่งตัวของ RISC-V ให้มีค่าเป็นศูนย์(zero register) ช่วยให้สามารถใช้งานคำสั่งเทียมเหล่านี้ได้อย่างมีประสิทธิภาพ คำสั่ง Load Upper Immediate (`lui`) และ Add Upper Immediate to PC (`auipc`) ช่วยให้คอมไพเลอร์และตัวเชื่อมโยงปรับที่อยู่สำหรับข้อมูลภายนอก และฟังก์ชันได้ง่ายขึ้น และการกระโดดสัมพัทธ์กับ PC (PC-relative branching) ช่วยให้ตัวเชื่อมโยงจัดการโค้ดที่ไม่ขึ้นกับตำแหน่งได้ง่ายขึ้น การมีรีจิสเตอร์จำนวนมากช่วยให้ข้อตกลงการเรียก (calling convention) ทำให้การเรียกและคืนค่าฟังก์ชันเร็วขึ้น โดยลดจำนวนการบันทึกและกู้คืนรีจิสเตอร์ (register spills and restores)

RISC-V นำเสนอชุดกลไกที่เรียบง่าย แต่ทรงประสิทธิภาพอย่างมีรสนิยม ซึ่งช่วยลดต้นทุน ปรับปรุงประสิทธิภาพ และทำให้การเขียนโปรแกรมง่ายขึ้น

3.8 เรียนรู้เพิ่มเติม

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

4 RV32M: การคูณและการหาร

วิลเลียมแห่งอ็อคแฮม (1287–1347) เป็น นักเทววิทยาชาวอังกฤษที่ส่งเสริมสิ่งที่ปัจจุบันเรียกว่า “มีดโกนของอ็อกแคม” “Occam's razor” ซึ่งเป็นการให้ความสำคัญกับความเรียบง่ายในวิธีการทางวิทยาศาสตร์



สิ่งต่างๆ ไม่ควรถูกเพิ่มทวีคูณเกินความจำเป็น

—วิลเลียมแห่งอ็อคแฮม, 1320

4.1 บทนำ

RV32M เพิ่มคำสั่งคูณและหารสำหรับจำนวนเต็มให้กับ RV32I รูปที่ 4.1 เป็นการแสดงภาพของชุดคำสั่งส่วนขยาย RV32M และรูปที่ 4.2 แสดงรายการรหัสดำเนินการ (opcodes) การหารนั้นตรงไปตรงมา มีสูตรดังนี้

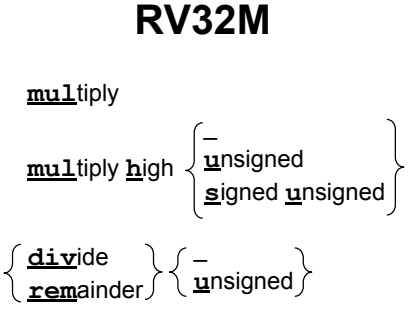
$$\text{ผลลัพท์} = (\text{ตัวตั้ง} - \text{เศษ}) \div \text{ตัวหาร}$$

หรืออีกนัยหนึ่ง

$$\text{ตัวตั้ง} = \text{ผลลัพธ์} \times \text{ตัวหาร} + \text{เศษ}$$

$$\text{เศษ} = \text{ตัวตั้ง} - (\text{ผลลัพธ์} \times \text{ตัวหาร})$$

RV32M มีคำสั่งการหารสำหรับทั้งจำนวนเต็ม ที่มีเครื่องหมายและไม่มีเครื่องหมาย: การหาร (`div`) และการหารแบบไม่มีเครื่องหมาย (`divu`) ซึ่งเก็บผลลัพธ์ลงในรีจิสเตอร์ปลายทาง ในบางกรณีที่โปรแกรมเมอร์ต้องการเศษ (remainder) แทนผลลัพธ์ RV32M มี



รูปที่ 4.1: แผนภาพของคำสั่ง RV32M

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

รูปที่ 4.2: แผนผังรหัสคำสั่งของ RV32M มีรูปแบบคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อคำสั่ง (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

```
# การคำนวณการหารแบบไม่มีเครื่องหมายของ a0 ด้วย 3 โดยใช้การคูณ
   0: aaaab2b7    lui   t0,0xaaaab  # t0 = 0xaaaaaaab
   4: aab28293    addi  t0,t0,-1365 #    = ~ 2^32 / 1.5
   8: 025535b3    mulhu a1,a0,t0    # a1 = ~ (a0 / 1.5)
   c: 0015d593    srli  a1,a1,0x1   # a1 = (a0 / 3)
```

รูปที่ 4.3: รหัส RV32M หารด้วยค่าคงที่โดยการคูณ การวิเคราะห์เชิงตัวเลขอย่างรอบคอบจำเป็นเพื่อแสดงว่าอัลกอริธึมนี้ทำงานได้สำหรับทุกตัวตั้ง (dividend) และสำหรับตัวหาร (divisors) บางตัว ขั้นตอนการแก้ไข อาจจะซับซ้อนกว่านี้ การพิสูจน์ความถูกต้องและอัลกอริทึมสำหรับการสร้างค่าผกผัน (reciprocals) และขั้นตอนการแก้ไขอยู่ใน [Granlund and Montgomery 1994].

คำสั่งเศษ (`rem`) และเศษแบบไม่มีเครื่องหมาย (`remu`) ซึ่งเขียนเศษลงในรีจิสเตอร์แทนผลลัพธ์

สมการการคูณนั้นง่ายกว่า:

$$\text{ผลคูณ} = \text{ตัวถูกคูณ} \times \text{ตัวคูณ}$$

การคูณซับซ้อนกว่าการหาร เพราะขนาดของผลคูณ เท่ากับผลรวมของขนาดของตัวคูณและตัวถูกคูณ การคูณจำนวน 32 บิตสองจำนวนจะได้ผลคูณ 64 บิต เพื่อให้ได้ผลคูณ 64 บิตที่มีเครื่องหมายหรือไม่มีเครื่องหมายที่ถูกต้อง RISC-V มีคำสั่งการคูณ 4 คำสั่ง —ใช้คำสั่ง `mul` เพื่อให้ได้ผลคูณของจำนวนเต็ม 32 บิตซึ่งเป็น 32 บิตล่างของผลคูณเต็ม —ใช้คำสั่ง `mulh` เพื่อให้ได้ 32 บิตบนของผลคูณ 64 บิต เมื่อตัวคูณทั้งสองมีเครื่องหมาย (signed) —ใช้คำสั้ง `mulhu` เมื่อถ้าตัวคูณทั้งสองไม่มีเครื่องหมาย และใช้คำสั่ง `mulhsu` เมื่อตัวคูณหนึ่งมีเครื่องหมายและอีกตัวไม่มีเครื่องหมาย เนื่องจากการเขียนผลคูณ 64 บิตลงในรีจิสเตอร์ 32 บิตสองตัวในคำสั่งเดียว จะทำให้ฮาร์ดแวร์ซับซ้อน RV32M จึงกำหนดให้ใช้สองคำสั่งการคูณเพื่อสร้างผลคูณ 64 บิต

คำสั่ง `srl` สามารถใช้สำหรับ การ คูณ แบบไม่มี เครื่องหมาย ด้วย 2^i ตัวอย่าง เช่น หาก a2 = 16 (2^4) คำ สั่ง `srli t2,a1,4` จะให้

คำสั่ง `sll` สามารถทำการคูณทั้งแบบมีเครื่องหมายและแบบไม่มีเครื่องหมายด้วย 2^i ตัวอย่าง เช่น ถ้า a2 = 16 (2^4) แล้ว `slli t2,a1,4` ได้ผลลัพท์เดียวกับ `mul t2,a1,a2`

สำหรับไมโครโปรเซสเซอร์หลายตัว การหารจำนวนเต็ม (integer division) เป็นการดำเนินการที่ค่อนข้างช้า ดังที่กล่าวไว้ข้างต้น การเลื่อนบิตไปทางขวา (right shifts) สามารถแทนที่การหารแบบไม่มีเครื่องหมาย (unsigned division) ด้วยกำลังของ 2 ได้



ประสิทธิภาพ

ปรากฏว่าการหารด้วยค่าคงที่อื่น ๆ ก็สามารถปรับให้เหมาะสมได้เช่นกัน โดยการคูณด้วยค่าผกผันโดยประมาณ (approximate reciprocal) แล้วใช้ขั้นตอนการแก้ไขกับครึ่งบนของผลคูณ ตัวอย่างเช่น รูปที่ 4.3 แสดงโค้ดสำหรับการหารแบบไม่มีเครื่องหมายด้วย 3

สำหรับโปรเซสเซอร์เกือบทั้งหมด การคูณจะช้ากว่าการเลื่อนบิต (shifts) หรือการบวก (adds) และการหารจะช้ากว่าการคูณมาก

อะไรที่ต่างกัน? ARM-32 มีคำสั่งการคูณแต่ไม่มีการหาร คำสั่งหารไม่ได้กลายเป็นสิ่งบังคับจนกระทั่งปี 2005 ซึ่งเกือบ 20 ปีหลังจากโปรเซสเซอร์ ARM ตัวแรก MIPS-32 ใช้รีจิสเตอร์พิเศษ (HI และ LO) เป็นรีจิสเตอร์ปลายทางเพียงอย่างเดียวสำหรับคำสั่งคูณและหาร แม้ว่าการออกแบบนี้จะลดความซับซ้อนของการใช้งาน MIPS ในยุคแรก ๆ แต่ต้องใช้คำสั่งย้าย (move) เพิ่มเติมเพื่อใช้ผลลัพธ์ของการคูณหรือหาร ซึ่งอาจลดประสิทธิภาพลง รีจิสเตอร์ HI และ LO ยังเพิ่มสถานะทางสถาปัตยกรรม ทำให้การสลับระหว่างงานช้าลงเล็กน้อย

■ ***รายละเอียดเพิ่มเติม: คำสั่ง*** `mulh` ***และ*** `mulhu` ***สามารถใช้ตรวจสอบการล้น (overflow) ในการคูณได้***

ไม่ตรวจสอบการล้น (overflow) เมื่อใช้คำสั่ง `mul` สำหรับการคูณแบบไม่มีเครื่องหมาย หากผลลัพธ์ของคำสั่ง `mulhu` เป็นศูนย์ ในทำนองเดียวกัน จะไม่ตรวจสอบการล้นเมื่อใช้คำสั่ง `mul` สำหรับการคูณแบบมีเครื่องหมาย หากบิตทั้งหมดในผลลัพธ์ของคำสั่ง `mulh` ตรงกับบิตเครื่องหมายของผลลัพธ์ของคำสั่ง `mul` นั่นคือ เท่ากับ 0 หากเป็นบวก หรือ ffff ffff$_{hex}$ หากเป็นลบ

■ ***รายละเอียดเพิ่มเติม: การตรวจสอบการหารด้วยศูนย์ก็ทำได้ง่าย***

เพียงเพิ่มการทดสอบด้วยคำสั่ง `beqz` สำหรับตัวหารก่อนการหาร RV32I ไม่สร้างข้อยกเว้น (trap) เมื่อหารด้วยศูนย์ เพราะโปรแกรมส่วนน้อยที่ต้องการพฤติกรรมนี้ และโปรแกรมที่ต้องการสามารถตรวจสอบศูนย์ได้ง่ายๆ ในซอฟต์แวร์ แน่นอนว่า การหารด้วยค่าคงที่ไม่จำเป็นต้องมีการตรวจสอบ

■ ***รายละเอียดเพิ่มเติม: คำสั่ง*** `mulhsu` ***มีประโยชน์สำหรับการคูณแบบมีเครื่องหมายมัลติเวิร์ด***

`mulhsu` สร้างครึ่งบนของผลคูณเมื่อตัวคูณมีเครื่องหมาย และตัวถูกคูณไม่มีเครื่องหมาย ขั้นตอนนี้ใช้เป็นขั้นตอนย่อยของการคูณ แบบมีเครื่องหมายมัลติเวิร์ด (multi-word) เมื่อคูณเวิร์ด (word) ที่มีนัยสำคัญที่สุดของตัวคูณ (ที่มีบิตเครื่องหมาย) กับเวิร์ด ที่มีนัยสำคัญน้อยกว่า (ที่ไม่มีเครื่องหมาย) คำสั่งนี้ช่วยปรับปรุงประสิทธิภาพของการคูณประมาณ 15%

4.2 ข้อสังเกตสรุป

ส่วนประกอบที่ถูกที่สุด รวดเร็วที่สุด และน่าเชื่อถือที่สุด คือส่วนประกอบที่ไม่มีอยู่จริง
—ซี. กอร์ดอน แบล สถาปนิกของมินิคอมพิวเตอร์ที่มีชื่อเสียง



ค่าใช้จ่าย

เพื่อให้ได้โปรเซสเซอร์ RISC-V ที่เล็กที่สุดสำหรับแอปพลิเคชันฝังตัว การคูณและการหารเป็นส่วนหนึ่งของส่วนขยายมาตรฐานเสริมตัวแรกของ RISC-V อย่างไรก็ตาม โปรเซสเซอร์ RISC-V จำนวนมากจะรวม RV32M ไว้ด้วย

4.3 เรียนรู้เพิ่มเติม

T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN Notices*, volume 29, pages 61--72. ACM, 1994.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

5 Zfh, RV32F และ RV32D: จุดลอยตัวแบบความแม่นยำครึ่งหนึ่ง, เดี่ยว, และคู่

การบรรลุถึงความสมบูรณ์แบบนั้น ไม่ใช่ไม่มีอะไรเพิ่มเติมอีกต่อไป หากต่อเมื่อ ไม่มีอะไรให้ตัดออกอีกแล้ว

—อองตวน เดอ แซงเต็กซูเปรี, *แตร์ เด ฮอมเมส*, 1939

อองตวน เดอ แซงเต็กซูเปรี (1900–1944) เป็นนักเขียนและนักบินชาวฝรั่งเศส เป็นที่รู้จักจากหนังสือ เจ้าชายน้อย (The Little Prince)



5.1 บทนำ

แม้ว่า RV32F และ RV32D จะเป็นส่วนขยายชุดคำสั่งที่แยกจากกัน และเป็นทางเลือก แต่ทั้งสองมักถูกรวมไว้ด้วยกัน หากโปรเซสเซอร์มีอย่างน้อย RV32F ก็สามารถรวม Zfh ได้ด้วย เนื่องจากมีคำสั่งจุดลอยตัวความแม่นยำครึ่งหนึ่ง (half-precision, 16 บิต), ความแม่นยำเท่าเดี่ยว (single-precision, 32 บิต), และความแม่นยำคู่ (double-precision, 64 บิต) สำหรับเกือบทุกคำสั่งจุดลอยตัว เพื่อความกระชับ เราจึงนำเสนอทั้งหมดในบทเดียว รูปที่ 5.1 เป็นการแสดงผลแบบกราฟิกของชุดคำสั่งส่วนขยาย Zfh, RV32F และ RV32D ในตอนท้ายของบท รูปที่ 5.13 แสดงรายการรหัสดำเนินการ (opcodes) ของ Zfh รูปที่ 5.11 แสดงรายการรหัสดำเนินการของ RV32F, และรูปที่ 5.12 แสดงรายการรหัสดำเนินการของ RV32D เช่นเดียวกับสถาปัตยกรรมชุดคำสั่งสมัยใหม่เกือบทุกตัว RISC-V ปฏิบัติตามมาตรฐานจุดลอยตัว IEEE 754-2019 [IEEE Standards Committee 2019]

ความแม่นยำครึ่งหนึ่งใช้ชื่อ ZfH เพราะถูกเพิ่มเข้ามาหลังจากที่ RISC-V เริ่มใช้การตั้งชื่อแบบตัวอักษร Z

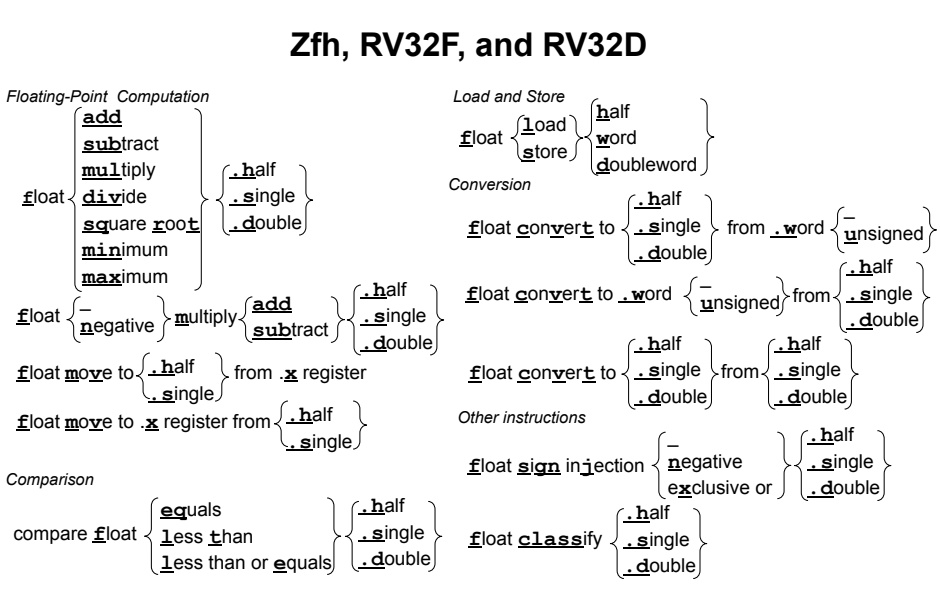
5.2 รีจิสเตอร์จุดลอยตัว



ประสิทธิภาพ

Zfh, RV32F และ RV32D ใช้รีจิสเตอร์ `f` จำนวน 32 ตัว แยกต่างหาก แทนการใช้รีจิสเตอร์ `x` เหตุผลหลักสำหรับการมีรีจิสเตอร์สองชุดนี้คือ โปรเซสเซอร์สามารถเพิ่มประสิทธิภาพได้โดยการเพิ่มความจุและแบนด์วิดท์ของรีจิสเตอร์เป็นสองเท่าด้วยการมีรีจิสเตอร์สองชุด โดยไม่ต้องเพิ่มพื้นที่สำหรับตัวระบุรีจิสเตอร์ในรูปแบบคำสั่ง RISC-V ที่มีข้อจำกัดด้านขนาด ผลกระทบหลักต่อชุดคำสั่งคือการเพิ่มคำสั่งใหม่เพื่อโหลดและบันทึกข้อมูลในรีจิสเตอร์ `f` และเพื่อถ่ายโอนข้อมูลระหว่างรีจิสเตอร์ `x` และรีจิสเตอร์ `f` รูปที่ 5.2 แสดงรายการรีจิสเตอร์ของ Zfh, RV32D และ RV32F พร้อมชื่อ ตามที่กำหนดโดย RISC-V ABI

หากโปรเซสเซอร์มีทั้ง RV32F และ RV32D ข้อมูลความแม่นยำเดี่ยว จะใช้เพียง 32 บิตล่างของรีจิสเตอร์ `f` ในทำนองเดียวกัน ความแม่นยำครึ่งหนึ่งจะใช้เพียง 16 บิตล่างของรีจิ



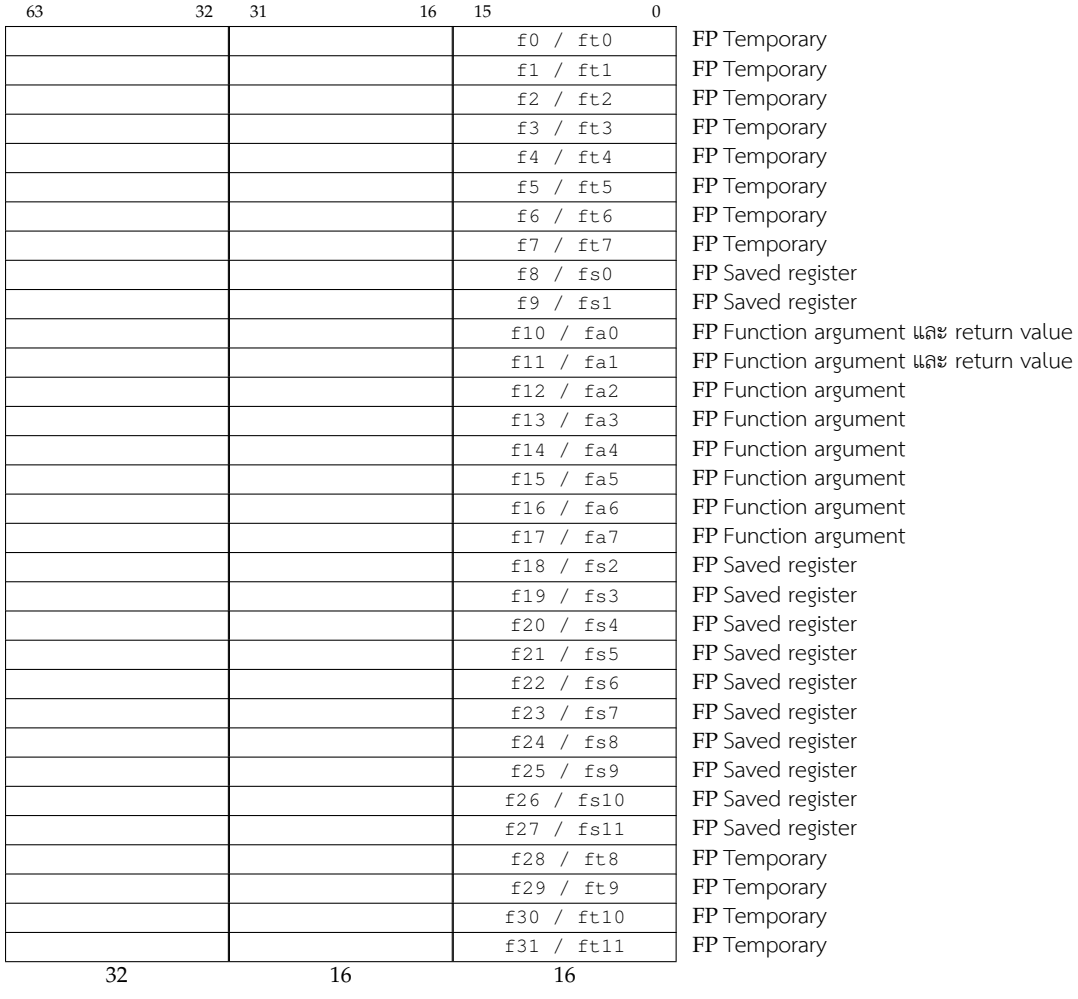
รูปที่ 5.1: แผนภาพของคำสั่ง Zfh, RV32F และ RV32D

สเตอร์ `f` ต่างจาก x0 ใน RV32I ที่ถูกกำหนดให้มีค่าคงที่เป็น 0 รีจิสเตอร์ `f0` *ไม่*ถูกกำหนดให้เป็น 0 แต่เป็นรีจิสเตอร์ที่เปลี่ยนแปลงได้เช่นเดียวกับรีจิสเตอร์ `f` อื่นๆ อีก 31 ตัว

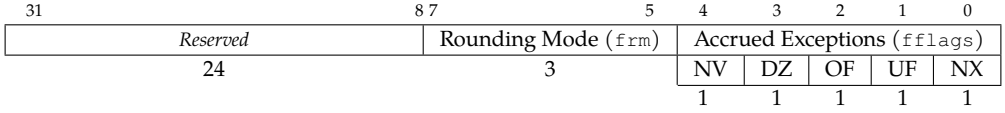
มาตรฐาน IEEE 754-2019เสนอ วิธี การ ปัดเศษ (rounding) การ คำนวณ จุด ลอยตัว หลายวิธี ซึ่งช่วยในการกำหนดขอบเขตข้อผิดพลาดและในการเขียนไลบรารีเชิงตัวเลข วิธีที่แม่นยำและใช้กันทั่วไปมากที่สุดคือการปัดไปยังจำนวนที่ใกล้ที่สุดและเป็นเลขคู่ (RNE: Round to Nearest Even) โหมดการปัดเศษถูกกำหนดในรีจิสเตอร์ควบคุมและสถานะจุดลอยตัว `fcsr` รูปที่ 5.3 แสดงรายละเอียดของ `fcsr` และระบุตัวเลือกการปัดเศษ นอกจากนี้ยังเก็บแฟล็กข้อยกเว้นที่สะสมไว้ตามที่มาตรฐานกำหนด

อะไรที่แตกต่าง? ทั้ง ARM-32 และ MIPS-32 มีรีจิสเตอร์จุดลอยตัวแบบความแม่นยำเดี่ยว 32 ตัว แต่มีรีจิสเตอร์แบบความแม่นยำคู่เพียง 16 ตัว โดยทั้งคู่จะจับคู่รีจิสเตอร์ความแม่นยำเดี่ยวสองตัวให้เป็นส่วนซ้ายและขวาของรีจิสเตอร์ความแม่นยำคู่ที่มีขนาด 32 บิตต่อข้าง ในทางกลับกัน การคำนวณจุดลอยตัวของ x86-32 ไม่มีรีจิสเตอร์เลย แต่ใช้สแต็ก (stack) แทน รายการในสแต็กมีความกว้าง 80 บิตเพื่อเพิ่มความแม่นยำ ดังนั้นการโหลดจะแปลงตัวถูกดำเนินการขนาด 32 บิตหรือ 64 บิตเป็น 80 บิต และในทางกลับกันสำหรับการบันทึก ต่อมา x86-32 รุ่นถัดไปได้เพิ่มรีจิสเตอร์จุดลอยตัวแบบดั้งเดิม 8 ตัวที่มีขนาด 64 บิตพร้อมคำสั่งที่เกี่ยวข้อง (หัวข้อ 8.12 ระบุว่า x86 ได้ปรับปรุงคำสั่งจุดลอยตัวเพิ่มเติมอีกสองครั้ง โดยแต่ละครั้งเพิ่มความกว้างและจำนวนรีจิสเตอร์เป็นสองเท่า) ต่างจาก RV32FD และ MIPS-32 ทั้ง ARM-32 และ x86-32 ไม่มีคำสั่งสำหรับการย้ายข้อมูลโดยตรงระหว่างรีจิสเตอร์จุดลอยตัวและรีจิสเตอร์จำนวนเต็ม วิธีเดียวคือต้องบันทึกรีจิสเตอร์จุดลอยตัวลงในหน่วยความจำ แล้วโหลดจากหน่วยความจำไปยังรีจิสเตอร์จำนวนเต็ม และในทางกลับกัน

การมีรีจิสเตอร์ความแม่นยำคู่เพียง 16 ตัวเป็นข้อผิดพลาดที่ร้ายแรงที่สุดในสถาปัตยกรรมชุดคำสั่ง (ISA) ของ MIPS ตามที่ John Mashey หนึ่งในสถาปนิกของ MIPS กล่าวไว้



รูปที่ 5.2: รีจิสเตอร์จุดลอยตัวของ Zfh, RV32F, และ RV32D รีจิสเตอร์ความแม่นยำเดี่ยวจะใช้ครึ่งขวาสุดของรีจิสเตอร์ความแม่นยำคู่ 32 ตัว และความแม่นยำครึ่งหนึ่งจะใช้ครึ่งขวาสุดของรีจิสเตอร์ความแม่นยำเดี่ยว บทที่ 3 อธิบายเกี่ยวกับการเรียกใช้งานตามข้อตกลง (calling convention) ของ RISC-V สำหรับรีจิสเตอร์จุดลอยตัว รวมถึงเหตุผลที่อยู่เบื้องหลังรีจิสเตอร์จุดลอยตัว (FP registers: fa0–fa7) รีจิสเตอร์จุดลอยตัวที่บันทึก (FP Saved registers: fs0–fs11) และชั่วคราวจุดลอยตัว (FP Temporaries: ft0–ft11) (ตารางที่ 20.1 ใน [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)



รูปที่ 5.3: รีจิสเตอร์ควบคุมและสถานะจุดลอยตัว รีจิสเตอร์นี้เก็บโหมดการปัดเศษ และแฟล็กข้อยกเว้น โหมดการปัดเศษประกอบด้วย: ปัดไปยังจำนวนที่ใกล้ที่สุดและเลือกเลขคู่เมื่อเสมอกัน (rte, 000 ใน `frm`); ปัดเศษไปยังศูนย์ (rtz, 001); ปัดลงไปยัง $-\infty$ (rdn, 010); ปัดขึ้นไปยัง $+\infty$ (rup, 011); และปัดไปยังจำนวนที่ใกล้ที่สุดและเลือกค่าที่มีขนาดใหญ่ที่สุดเมื่อเสมอกัน (rmm, 100) แฟล็กข้อยกเว้นที่สะสมไว้ 5 ตัว ระบุเงื่อนไขข้อยกเว้นที่เกิดขึ้นจากการคำนวณจุดลอยตัวตั้งแต่ครั้งสุดท้ายที่ถูกรีเซ็ตโดยซอฟต์แวร์: NV คือ คือการดำเนินการไม่ถูกต้อง (Invalid Operation); DZ คือการหารด้วยศูนย์ (Divide by Zero); OF คือการล้นหลัก (Overflow); UF คือต่ำกว่าค่าต่ำสุด (Underflow); และ NX คือไม่แม่นยำ (Inexact) (รูปที่ 8.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

■ ***รายละเอียดเพิ่มเติม: RV32FD อนุญาตให้กำหนดโหมดการปัดเศษสำหรับแต่ละคำสั่งได้***

เรียกว่า *การปัดเศษแบบคงที่* (static rounding) ซึ่งช่วยเพิ่มประสิทธิภาพเมื่อต้องการเปลี่ยนโหมดการปัดเศษสำหรับคำสั่งเดียวเท่านั้น โดยค่าเริ่มต้นจะใช้โหมดการปัดเศษแบบไดนามิกในคำสั่ง `fcsr` การปัดเศษแบบคงที่ ที่ถูกระบุเป็นอาร์กิวเมนต์สุดท้ายที่เป็นทางเลือก เช่น `fadd.s ft0, ft1, ft2, rtz` จะปัดเศษไปยังศูนย์ โดยไม่ขึ้นกับคำสั่ง `fcsr` คำบรรยายของรูปที่ 5.3 แสดงชื่อของโหมดการปัดเศษ

5.3 การโหลด การบันทึก และการคำนวณจุดลอยตัว

การถ่ายโอนข้อมูลแบบความแม่นยำเดี่ยวใช้ตัวย่อ “w” แทน “s” เพราะเป็นข้อมูลขนาด 32 บิตเท่านั้น

RISC-V มีคำสั่งโหลด 3 คำสั่ง (`flh`, `flw`, `fld`) และคำสั่งบันทึก 3 คำสั่ง (`fsh`, `fsw`, `fsd`) สำหรับ Zfh, RV32F และ RV32D คำสั่งเหล่านี้ใช้โหมดการกำหนดที่อยู่และรูปแบบคำสั่งเดียวกันกับคำสั่ง `lw` และ `sw`

ต่างจากการคำนวณจำนวนเต็ม ขนาดของผลคูณจากการคูณจุดลอยตัวจะเท่ากับขนาดของตัวถูกดำเนินการ นอกจากนี้ Zfh, RV32F และ RF32D ไม่มีคำสั่งสำหรับการหาหารเศษส่วนจุดลอยตัว

นอกเหนือจากการดำเนินการทางคณิตศาสตร์มาตรฐานสี่คำสั่ง (`fadd.h`, `fadd.s`, `fadd.d`, `fsub.h`, `fsub.s`, `fsub.d`, `fmul.h`, `fmul.s`, `fmul.d`, `fdiv.h`, `fdiv.s`, `fdiv.d`) แล้ว Zfh, RV32F และ RV32D ยังรวมถึงการหาค่ารากที่สอง (`fsqrt.h`, `fsqrt.s`, `fsqrt.d`) และมีคำสั่งหาค่าต่ำสุดและค่าสูงสุด (`fmin.h`, `fmin.s`, `fmin.d`, `fmax.h`, `fmax.s`, `fmax.d`) ซึ่งเขียนค่าที่เล็กกว่าหรือใหญ่กว่าจากคู่ตัวถูกดำเนินการโดยไม่ต้องใช้คำสั่งกระโดดแบบมีเงื่อนไข



ประสิทธิภาพ

อัลกอริธึมจุดลอยตัวจำนวนมาก เช่น การคูณเมทริกซ์ จะดำเนินการคูณแล้วตามด้วยการบวกหรือการลบในทันที ดังนั้น RISC-V จึงมีคำสั่งที่คูณตัวถูกดำเนินการสองตัวแล้วบวก (`fmadd.h`, `fmadd.s`, `fmadd.d`) หรือลบ (`fmsub.h`, `fmsub.s`, `fmsub.d`) ตัวถูกดำเนินการตัวที่สามกับผลคูณนั้นก่อนเขียนผลรวม นอกจากนี้ยังมีคำสั่งที่เปลี่ยนเครื่องหมายของผลคูณ (negate the product) ก่อนการบวกหรือลบตัวถูกดำเนินการตัว

To	From				
	32b signed integer (w)	32b unsigned integer (wu)	16b floating point (h)	32b floating point (s)	64b floating point (d)
32b signed int (w)	–	–	fcvt.w.h	fcvt.w.s	fcvt.w.d
32b unsigned int (wu)	–	–	fcvt.wu.h	fcvt.wu.s	fcvt.wu.d
16b fl. pt. (h)	fcvt.h.w	fcvt.h.wu	–	fcvt.h.s	fcvt.h.d
32b fl. pt. (s)	fcvt.s.w	fcvt.s.wu	fcvt.s.h	–	fcvt.s.d
64b fl. pt. (d)	fcvt.d.w	fcvt.d.wu	fcvt.d.h	fcvt.d.s	–

รูปที่ 5.4: คำสั่งแปลงข้อมูลของ Zfh, RV32F และ RV32D คอลัมน์แสดงประเภทข้อมูลต้นทาง และแถวแสดงประเภทข้อมูลปลายทางที่แปลงแล้ว

ที่สาม: `fnmadd.h`, `fnmadd.s`, `fnmadd.d`, `fnmsub.h`, `fnmsub.s`, `fnmsub.d` คำสั่งคูณ-บวกแบบ*ผสม* (fused) เหล่านี้ถูกกำหนดโดยมาตรฐาน IEEE 754-2019 เพื่อความแม่นยำที่สูงขึ้น โดยปัดเศษเพียงครั้งเดียว (หลังการบวก) แทนที่จะปัดสองครั้ง (หลังการคูณแล้วจึงปัดหลังการบวก) การข้ามการปัดเศษระหว่างขั้นตอน มีผลอย่างมากเมื่อผลคูณและตัวบวกมีขนาดใกล้เคียงกันแต่มีเครื่องหมายตรงข้าม ซึ่งทำให้บิตส่วนใหญ่ของ mantissa หักล้างกันในการลบ คำสั่งเหล่านี้ต้องใช้รูปแบบคำสั่งใหม่ที่เรียกว่า R4 เพื่อระบุรีจิสเตอร์ 4 ตัว ซึ่งเป็นรูปแบบที่ดัดแปลงจากรูปแบบ R รูปที่ 5.11 และ 5.12 แสดงรูปแบบ R4

แทนที่จะมีคำสั่งกระโดดตามเงื่อนไขจุดลอยตัว Zfh, RV32F และ RV32D มีคำสั่งเปรียบเทียบที่ตั้งค่ารีจิสเตอร์จำนวนเต็มเป็น 1 หรือ 0 ตามผลการเปรียบเทียบของรีจิสเตอร์จุดลอยตัวสองตัว เท่ากัน (`feq.h`, `feq.s`, `feq.d`), น้อยกว่า (`flt.h`, `flt.s`, `flt.d`), น้อยกว่าหรือเท่ากับ (`fle.h`, `fle.s`, `fle.d`) คำสั่งเหล่านี้ ช่วยให้คำสั่งกระโดดจำนวนเต็มสามารถกระโดดตามเงื่อนไขจุดลอยตัวได้ ตัวอย่างเช่น โค้ดนี้จะกระโดดไปยัง `Exit` หาก `f1 < f2`

```
flt.d x5, f1, f2   # x5 = 1 ถ้า f1 < f2; มิฉะนั้น x5 = 0
bne   x5, x0, Exit # ถ้า x5 != 0, ข้ามไปที่ Exit
```

5.4 การย้ายและแปลงข้อมูลจุดลอยตัว

Zfh, RV32F และ RV32D มีคำสั่งที่รองรับการแปลงข้อมูลเป็นประโยชน์ ระหว่างจำนวนเต็มแบบมีเครื่องหมาย 32 บิต, จำนวนเต็มไม่มีเครื่องหมาย 32 บิต, จุดลอยตัวในระดับความแม่นยำครึ่งหนึ่ง (16 บิต), ความแม่นยำเดี่ยว (32 บิต), และความแม่นยำคู่ (64 บิต) โดยมีคำสั่งทั้งหมด 10 คำสั่งสำหรับการแปลงนี้ รูปที่ 5.4 แสดงรายการคำสั่งทั้ง 10 คำสั่ง โดยแบ่งตามประเภทข้อมูลต้นทางและประเภทข้อมูลปลายทางที่ถูกแปลง

นอกจากนี้ RV32F ยังมีคำสั่งสำหรับการโอนย้ายข้อมูลระหว่างรีจิสเตอร์จุดลอยตัว `f` และรีจิสเตอร์จำนวนเต็ม `x` ซึ่งรวมถึงคำสั่ง (`fmv.x.h`, `fmv.x.w`) สำหรับย้ายจาก `f` ไป `x` และในทางกลับกัน คำสั่ง (`fmv.h.x`, `fmv.w.x`) สำหรับย้ายจาก `x` ไป `f`

5.5 คำสั่งจุดลอยตัวสำหรับไลบรารีคณิตศาสตร์

Zfh, RV32F และ RV32D มีคำสั่งพิเศษ ที่ช่วยในการพัฒนาไลบรารีคณิตศาสตร์และคำสั่งเทียม (pseudoinstructions) ที่เป็นประโยชน์ (มาตรฐาน IEEE 754 ต้องการวิธีการคัดลอกและจัดการเครื่องหมาย รวมถึงการจำแนกข้อมูลจุดลอยตัว ซึ่งเป็นแรงบันดาลใจให้เกิดคำสั่งเหล่านี้)

คำสั่งแรกคือคำสั่ง *ฉีดเครื่องหมาย (sign-injection)* ซึ่งคัดลอกทุกอย่างจากรีจิสเตอร์ต้นทางตัวแรกยกเว้นบิตเครื่องหมาย โดยค่าของบิตเครื่องหมายขึ้นอยู่กับคำสั่งดังนี้:

1. (ฉีดเครื่องหมายจุดลอยตัว `fsgnj.h`, `fsgnj.s`, `fsgnj.d`) บิตเครื่องหมายของผลลัพท์คือบิตเครื่องหมายของ rs2
2. (ฉีดเครื่องหมายจุดลอยตัวแบบลบ `fsgnjn.h`, `fsgnjn.s`, `fsgnjn.d`) บิตเครื่องหมายของผลลัพท์ตรงข้ามกับบิตเครื่องหมายของ rs2
3. (ฉีดเครื่องหมายจุดลอยตัวแบบ XOR `fsgnjx.h`, `fsgnjx.s`, `fsgnjx.d`) บิตเครื่องหมายเป็นผลการดำเนินการ XOR ระหว่างบิตเครื่องหมายของ rs1 และ rs2

นอกจากช่วยในการจัดการเครื่องหมายในไลบรารีคณิตศาสตร์แล้ว คำสั่งฉีดเครื่องหมายยังให้คำสั่งเทียมจุดลอยตัวที่นิยมใช้ สามคำสั่ง (ดูรูปที่ 3.4 ในหน้า 43)

1. คัดลอกรีจิสเตอร์จุดลอยตัว:
   `fmv.h rd,rs` คือคำสั่ง `fsgnj.h rd,rs,rs`,
   `fmv.s rd,rs` คือคำสั่ง `fsgnj.s rd,rs,rs`, และ
   `fmv.d rd,rs` คือคำสั่ง `fsgnj.d rd,rs,rs`.
2. แปลงเครื่องหมายเป็นลบ
   `fneg.h rd,rs` แปลงเป็น `fsgnjn.h rd,rs,rs`,
   `fneg.s rd,rs` แปลงเป็น `fsgnjn.s rd,rs,rs`, และ
   `fneg.d rd,rs` แปลงเป็น `fsgnjn.d rd,rs,rs`,
3. ค่าสัมบูรณ์ (เนื่องจาก $0 \oplus 0 = 0$ และ $1 \oplus 1 = 0$)
   `fabs.h rd,rs` กลายเป็น `fsgnjx.h rd,rs,rs`
   `fabs.s rd,rs` กลายเป็น `fsgnjx.s rd,rs,rs`
   `fabs.d rd,rs` กลายเป็น `fsgnjx.d rd,rs,rs`

คำสั่งจุดลอยตัวพิเศษคำสั่งที่สองคือ คำสั่งจำแนกประเภท (`fclass.h`, `fclass.s`, `fclass.d`) คำสั่งจำแนกประเภทเป็นเครื่องมือที่ยอดเยี่ยมสำหรับไลบรารีคณิตศาสตร์ โดยจะตรวจสอบตัวถูกดำเนินการต้นทางเพื่อดูว่าตรงกับคุณสมบัติจุดลอยตัวใดใน 10 คุณสมบัติที่ใช้ได้ (ดูตารางด้านล่าง) แล้วเขียนมาสก์ลงใน 10 บิตล่างของรีจิสเตอร์จำนวนเต็มปลายทาง โดยจะมีเพียงหนึ่งในสิบบิตที่ถูกตั้งค่าเป็น 1 ส่วนที่เหลือเป็น 0 รายละเอียดดังตารางด้านล่าง:

```
void daxpy(size_t n, double a, const double x[], double y[])
{
  for (size_t i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```

รูปที่ 5.5: โปรแกรม DAXPY ซึ่งเน้นการคำนวณจำนวนจุดลอยตัว เขียนด้วยภาษา C รูปแบบของเราถูกทำให้เรียบง่ายจาก DAXPY มาตรฐาน ซึ่งรวมถึงพารามิเตอร์ stride และส่งผ่านอาร์กิวเมนต์โดยการอ้างอิงเพื่อให้เข้ากันได้กับ Fortran

x[rd] bit	ความหมาย
0	f[*rs1*] จำนวนอนันต์ลบ $-\infty$
1	f[*rs1*] จำนวนปกติที่เป็นลบ
2	f[*rs1*] จำนวนย่อยที่เป็นลบ
3	f[*rs1*] เป็น -0
4	f[*rs1*] เป็น $+0$
5	f[*rs1*] จำนวนย่อยที่เป็นบวก
6	f[*rs1*] จำนวนปกติที่เป็นบวก
7	f[*rs1*] จำนวนอนันต์บวก $+\infty$
8	f[*rs1*] เป็น NaN แบบส่งสัญญาณ
9	f[*rs1*] เป็น NaN แบบไม่มีสัญญาณ

5.6 ส่วนขยายมาตรฐาน “Q” สำหรับจุดลอยตัวความแม่นยำควอด (Quad-Precision Floating-Point)

ส่วนขยาย Q เพิ่มคำสั่งสำหรับ จำนวนจุดลอยตัว แบบไบนารีความแม่นยำควอดขนาด 128 บิตที่สอดคล้องกับมาตรฐานการคำนวณ IEEE 754-2019 รีจิสเตอร์ จำนวนจุดลอยตัว ได้รับการขยายเพื่อรองรับค่า จำนวนจุดลอยตัว แบบความแม่นยำเดี่ยว (single-precision), ความแม่นยำคู่ (double-precision) หรือ ความแม่นยำควอด (quad-precision) ส่วนขยาย สำหรับ จำนวน จุด ลอยตัว แบบ ไบนารี ความ แม่น ยำ ค วอด นี้ ต้องการ RV64IFD เนื่องจากการคำนวณจำนวนจุดลอยตัว ขนาด 128 บิต ต้องใช้ฮาร์ดแวร์จำนวนมาก เราไม่ทราบถึงการใช้งานฮาร์ดแวร์เชิงพาณิชย์ใดๆ สำหรับ RV64Q

5.7 การเปรียบเทียบ RV32FD, ARM-32, MIPS-32 และ x86-32 โดยใช้ DAXPY

เราจะทำการเปรียบเทียบแบบตัวต่อตัวโดยใช้ DAXPY เป็นเกณฑ์วัดประสิทธิภาพสำหรับจำนวนจุดลอยตัว (รูปที่ 5.5) โปรแกรมนี้คำนวณสมการ $Y = a \times X + Y$ ในรูปแบบ ความแม่นยำคู่ โดยที่ `X` และ `Y` เป็นเวกเตอร์ และ `a` เป็นสเกลาร์ รูปที่ 5.6 สรุปจำนวนคำสั่งและจำนวนไบต์ของโค้ด DAXPY สำหรับสี่สถาปัตยกรรมคำสั่ง โค้ดของแต่ละ ISA แสดงในรูป

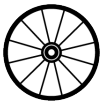
ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
Instructions	10	10	12	12	16	11	11
Per Loop	6	6	7	7	6	7	7
Bytes	40	28	48	32	50	44	28

รูปที่ 5.6: จำนวนคำสั่งและขนาดของโค้ดสำหรับ DAXPY ในสี่สถาปัตยกรรมชุดคำสั่ง รายการนี้แสดงจำนวนคำสั่งต่อลูปและจำนวนรวม บทที่ 7 อธิบายเกี่ยวกับ ARM Thumb-2, microMIPS และ RV32C

ชื่อ DAXPY มาจากสูตรการคำนวณ: Double-precision A times X Plus Y รูปแบบความแม่นยำเดี่ยวเรียกว่า เรียกว่า SAXPY

ที่ 5.7 ถึง 5.10

เช่นเดียวกับกรณีของการเรียงลำดับการแทรก (Insertion Sort) ในบทที่ 2 แม้ว่า RISC-V จะเน้นความเรียบง่าย เวอร์ชัน RISC-V มีจำนวนคำสั่งใกล้เคียงหรือน้อยกว่า และขนาดโค้ดของสถาปัตยกรรมต่างๆ ค่อนข้างใกล้เคียงกัน ในตัวอย่างนี้ คำสั่งเปรียบเทียบและดำเนินการกระโดดแบบมีเงื่อนไข ของ RISC-V ช่วยลดจำนวนคำสั่งได้มากเท่ากับโหมดการกำหนดที่อยู่ที่ซับซ้อน และคำสั่ง push และ pop ของ ARM-32 และ x86-32



ความเรียบง่าย



ประสิทธิภาพ

5.8 บทสรุป

น้อยแต่มาก

—คำกล่าวของ โรเบิร์ต บราวนิ่ง, ในปี 1855 และ โรงเรียนแนวคิดมินิมัลลิสต์ด้านสถาปัตยกรรม (Minimalist school of architecture) ได้คำนี้มาใช้เป็นหลักการในทศวรรษ 1980

มาตรฐาน IEEE 754-2019 สำหรับ จำนวน จุด ลอยตัว [IEEE Standards Committee 2019] กำหนด ประเภท ข้อมูล จำนวน จุด ลอยตัว, ความ แม่นยำ ใน การ คำนวณ และ การดำเนินการที่ต้องมี ความสำเร็จของมาตรฐานนี้ช่วยลดความยากในการย้ายโปรแกรมที่ใช้จำนวนจุดลอยตัว ไปยังแพลตฟอร์มต่าง ๆ และยังทำให้ชุดคำสั่งสำหรับ จำนวนจุดลอยตัว มีความเหมือนกันมากกว่าชุดคำสั่งอื่นๆ ในบทอื่นๆ

■ ***รายละเอียดเพิ่มเติม: การปรับปรุงมาตรฐาน IEEE 754***

มาตรฐาน IEEE 754-2008 ที่ได้รับการปรับปรุง (IEEE 754-2008) [IEEE Standards Committee 2008] ได้เพิ่มรูปแบบใหม่หลายรูปแบบนอกเหนือจากความแม่นยำเดี่ยว (binary32) และความแม่นยำคู่ (binary64) ของมาตรฐานเดิม รูปแบบใหม่สองรูปแบบคือความแม่นยำครึ่งหนึ่ง (binary16) และความแม่นยำแปดเท่า (binary256) ซึ่งได้อธิบายไว้ข้างต้น การเพิ่มที่น่าประหลาดใจในมาตรฐานที่ปรับปรุงคือ จำนวนจุดลอยตัวฐานสิบ (decimal floating point) ซึ่ง RISC-V ได้สงวนชุดคำสั่ง RV32L ไว้ชั่วคราว รูปแบบฐานสิบสามรูปแบบที่มีชื่อชัดเจนในตัวเองคือ *decimal32*, *decimal64* และ *decimal128* เช่นเดียวกับ RV64Q เราไม่ทราบว่ามีการใช้งานฮาร์ดแวร์เชิงพาณิชย์ใด ๆ สำหรับ จำนวนจุดลอยตัวฐานสิบ ในโปรเซสเซอร์ RISC-V IEEE 754-2019 ซึ่งเผยแพร่ในเดือนกรกฎาคม 2019 เป็นการปรับปรุงเล็กน้อยจาก IEEE 754-2008 โดยเน้นการชี้แจง การแก้ไขข้อบกพร่อง และแนะนำการดำเนินการใหม่

5.9 เรียนรู้เพิ่มเติม

IEEE Standards Committee. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008.

IEEE Standards Committee. 754-2019 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2019.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

```
# RV32FD (7 คำสั่ง in loop; 11 คำสั่ง/44 bytes total; 28 bytes RVC)
# a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], fa0 คือ a
  0: 02050463 beqz    a0,28           # ถ้า n == 0 กระโดดไปที่ Exit
  4: 00351513 slli    a0,a0,0x3       # a0 = n*8
  8: 00a60533 add     a0,a2,a0        # a0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
Loop:
  c: 0005b787 fld     fa5,0(a1)       # fa5 = x[]
 10: 00063707 fld     fa4,0(a2)       # fa4 = y[]
 14: 00860613 addi    a2,a2,8         # a2++ (เพิ่มตัวชี้ไปที่ y)
 18: 00858593 addi    a1,a1,8         # a1++ (เพิ่มตัวชี้ไปที่ x)
 1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 20: fef63c27 fsd     fa5,-8(a2)      # y[i] = a*x[i] + y[i]
 24: fea614e3 bne     a2,a0,c         # ถ้า i != n กระโดดไปที่ Loop
Exit:
 28: 00008067         ret             # คืนค่า
```

รูปที่ 5.7: โค้ด RV32D สำหรับ DAXPY ในรูปที่ 5.5 ที่อยู่หน่วยความจำในรูปแบบฐานสิบหกอยู่ทางด้านซ้าย รหัสภาษาเครื่องในฐานสิบหกอยู่ถัดมา ตามด้วยคำสั่งภาษาแอสเซมบลีและคำอธิบาย คำสั่งเปรียบเทียบและกระโดดแบบมีเงื่อนไข (compare-and-branch) ช่วยลดการใช้คำสั่งเปรียบเทียบสองคำสั่งในโค้ดของ ARM-32 และ x86-32

```
# ARM-32 (6 คำสั่ง in loop; 10 คำสั่ง/40 bytes total; 28 bytes Thumb-2)
# r0 คือ n, d0 คือ a, r1 เป็นตัวชี้ไปที่ x[0], r2 เป็นตัวชี้ไปที่ y[0]
  0: e3500000 cmp      r0, #0             # เปรียบเทียบ n to 0
  4: 0a000006 beq      24 <daxpy+0x24>    # ถ้า n == 0 กระโดดไปที่ Exit
  8: e0820180 add      r0, r2, r0, lsl #3 # r0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
Loop:
  c: ecb16b02 vldmia   r1!,{d6}           # d6 = x[i], เพิ่มตัวชี้ไปที่ x
 10: ed927b00 vldr     d7,[r2]            # d7 = y[i]
 14: ee067b00 vmla.f64 d7, d6, d0         # d7 = a*x[i] + y[i]
 18: eca27b02 vstmia   r2!, {d7}          # y[i] = a*x[i] + y[i], เพิ่มตัวชี้ไปที่ y
 1c: e1520000 cmp      r2, r0             # i vs. n
 20: 1afffff9 bne      c <daxpy+0xc>      # ถ้า i != n กระโดดไปที่ Loop
Exit:
 24: e12fff1e bx       lr                 # คืนค่า
```

รูปที่ 5.8: โค้ด ARM-32 สำหรับ DAXPY ในรูปที่ 5.5 โหมดการกำหนดที่อยู่แบบเพิ่มอัตโนมัติ (autoincrement addressing mode) ของ ARM-32 ช่วยลดการใช้สองคำสั่งเมื่อเทียบกับ RISC-V ต่างจาก Insertion Sort ไม่จำเป็นต้องใช้คำสั่ง push และ pop สำหรับรีจิสเตอร์ใน DAXPY บน ARM-32

```
  # MIPS-32 (7 คำสั่ง in loop; 12 คำสั่ง/48 bytes total; 32 bytes microMIPS)
  # a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], f12 คือ a
    0: 10800009 beqz   a0,28 <daxpy+0x28>  # ถ้า n = 0 กระโดดไปที่ Exit
    4: 000420c0 sll    a0,a0,0x3           # a0 = n*8  หน่วงเวลาการกระโดดตามเงื่อนไข
    8: 00c42021 addu   a0,a2,a0            # a0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
  Loop:
    c: 24c60008 addiu  a2,a2,8             # a2++ (เพิ่มตัวชี้ไปที่ y)
   10: d4a00000 ldc1   $f0,0(a1)           # f0 = x[i]
   14: 24a50008 addiu  a1,a1,8             # a1++ (เพิ่มตัวชี้ไปที่ x)
   18: d4c2fff8 ldc1   $f2,-8(a2)          # f2 = y[i]
   1c: 4c406021 madd.d $f0,$f2,$f12,$f0    # f0 = a*x[i] + y[i]
   20: 14c4fffa bne    a2,a0,c <daxpy+0xc> # ถ้า i != n กระโดดไปที่ Loop
   24: f4c0fff8 sdc1   $f0,-8(a2)          # y[i] = a*x[i] + y[i] (ช่องเสียบเติม)
  Exit:
   28: 03e00008 jr     ra                  # คืนค่า
   2c: 00000000 nop                        # ไม่ทำการหน่วงเวลา
```

รูปที่ 5.9: โค้ด MIPS-32 สำหรับ DAXPY ในรูปที่ 5.5 สองในสามช่องหน่วงการกระโดด (branch delay slots) ถูกเติมด้วยคำสั่งที่มีประโยชน์ ความสามารถในการตรวจสอบความเท่ากันระหว่างสองรีจิสเตอร์ช่วยลดการใช้คำสั่งเปรียบเทียบสองคำสั่งที่พบใน ARM-32 และ x86-32 การโหลด จำนวนจุดลอยตัว ไม่มีช่องหน่วง (delay slot) ซึ่งต่างจากการโหลดจำนวนเต็ม

```
# x86-32 (6 คำสั่ง in loop; 16 คำสั่ง/50 bytes total)
# eax คือ i, n อยู่ในหน่วยความจำที่ esp+0x8, a อยู่ในหน่วยความจำที่ esp+0xc
# ตัวชี้ไปที่ x[0] อยู่ในหน่วยความจำที่ esp+0x14
# ตัวชี้ไปที่ y[0] อยู่ในหน่วยความจำที่ esp+0x18
  0: 53                   push       ebx                       # บันทึก ebx
  1: 8b 4c 24 08          mov        ecx,[esp+0x8]             # ecx มีสำเนาของ n
  5: c5 fb 10 4c 24 0c vmovsd        xmm1,[esp+0xc]            # xmm1 มีสำเนาของ a
  b: 8b 5c 24 14          mov        ebx,[esp+0x14]            # ebx ชี้ไปที่ x[0]
  f: 8b 54 24 18          mov        edx,[esp+0x18]            # edx ชี้ไปที่ y[0]
 13: 85 c9                test       ecx,ecx                   # เปรียบเทียบ n to 0
 15: 74 19                je         30 <daxpy+0x30>           # ถ้า n = 0 กระโดดไปที่ Exit
 17: 31 c0                xor        eax,eax                   # i=0 (since $x^x==$0)
Loop:
 19: c5 fb 10 04 c3    vmovsd       xmm0,[ebx+eax*8]          # xmm0 = x[i]
 1e: c4 e2 f1 a9 04 c2 vfmadd213sd  xmm0,xmm1,[edx+eax*8]     # xmm0 = a*x[i] + y[i]
 24: c5 fb 11 04 c2    vmovsd       xmm0,xmm1,[edx+eax*8]     # y[i] = a*x[i] + y[i]
 29: 83 c0 01          add          eax,0x1                   # i++
 2c: 39 c1             cmp          ecx,eax                   # เปรียบเทียบ i vs n
 2e: 75 e9             jne          19 <daxpy+0x19>           # ถ้า i!=n กระโดดไปที่ Loop
Exit:
 30: 5b                pop          ebx                       # กู้คืน ebx
 31: c3                ret                                    # คืนค่า
```

รูปที่ 5.10: โค้ด x86-32 สำหรับ DAXPY ในรูปที่ 5.5 การขาดแคลนรีจิสเตอร์ใน x86-32 เห็นได้ชัดในตัวอย่างนี้ โดยมีสี่ตัวแปรที่จัดสรรในหน่วยความจำ ในขณะที่โค้ดของสถาปัตยกรรมอื่น ๆ ใช้รีจิสเตอร์ นอกจากนี้ยังแสดงถึงลักษณะเฉพาะของ x86-32 เช่น การเปรียบเทียบรีจิสเตอร์กับศูนย์ (`test ecx,ecx`) หรือการตั้งค่ารีจิสเตอร์เป็นศูนย์ (`xor eax,eax`)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0000111	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	S fsw
rs3	00	rs2	rs1	rm	rd	1000011	R4 fmadd.s
rs3	00	rs2	rs1	rm	rd	1000111	R4 fmsub.s
rs3	00	rs2	rs1	rm	rd	1001011	R4 fnmsub.s
rs3	00	rs2	rs1	rm	rd	1001111	R4 fnmadd.s
0000000		rs2	rs1	rm	rd	1010011	R fadd.s
0000100		rs2	rs1	rm	rd	1010011	R fsub.s
0001000		rs2	rs1	rm	rd	1010011	R fmul.s
0001100		rs2	rs1	rm	rd	1010011	R fdiv.s
0101100		00000	rs1	rm	rd	1010011	R fsqrt.s
0010000		rs2	rs1	000	rd	1010011	R fsgnj.s
0010000		rs2	rs1	001	rd	1010011	R fsgnjn.s
0010000		rs2	rs1	010	rd	1010011	R fsgnjx.s
0010100		rs2	rs1	000	rd	1010011	R fmin.s
0010100		rs2	rs1	001	rd	1010011	R fmax.s
1100000		00000	rs1	rm	rd	1010011	R fcvt.w.s
1100000		00001	rs1	rm	rd	1010011	R fcvt.wu.s
1110000		00000	rs1	000	rd	1010011	R fmv.x.w
1010000		rs2	rs1	010	rd	1010011	R feq.s
1010000		rs2	rs1	001	rd	1010011	R flt.s
1010000		rs2	rs1	000	rd	1010011	R fle.s
1110000		00000	rs1	001	rd	1010011	R fclass.s
1101000		00000	rs1	rm	rd	1010011	R fcvt.s.w
1101000		00001	rs1	rm	rd	1010011	R fcvt.s.wu
1111000		00000	rs1	000	rd	1010011	R fmv.w.x

รูปที่ 5.11: แผนผังรหัสดำเนินการของ RV32F มีรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง ความแตกต่างหลักในการเข้ารหัสระหว่างรูปนี้กับรูปถัดไปคือ บิตที่ 12 เป็น 0 สำหรับสองคำสั่งแรก และบิตที่ 25 เป็น 0 สำหรับคำสั่งที่เหลือ โดยทั้งสองบิตนี้เป็น 1 ใน RV32D (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มารูปนี้)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsd
rs3	01	rs2	rs1	rm	rd	1000011	R4 fmadd.d
rs3	01	rs2	rs1	rm	rd	1000111	R4 fmsub.d
rs3	01	rs2	rs1	rm	rd	1001011	R4 fnmsub.d
rs3	01	rs2	rs1	rm	rd	1001111	R4 fnmadd.d
0000001		rs2	rs1	rm	rd	1010011	R fadd.d
0000101		rs2	rs1	rm	rd	1010011	R fsub.d
0001001		rs2	rs1	rm	rd	1010011	R fmul.d
0001101		rs2	rs1	rm	rd	1010011	R fdiv.d
0101101		00000	rs1	rm	rd	1010011	R fsqrt.d
0010001		rs2	rs1	000	rd	1010011	R fsgnj.d
0010001		rs2	rs1	001	rd	1010011	R fsgnjn.d
0010001		rs2	rs1	010	rd	1010011	R fsgnjx.d
0010101		rs2	rs1	000	rd	1010011	R fmin.d
0010101		rs2	rs1	001	rd	1010011	R fmax.d
0100000		00001	rs1	rm	rd	1010011	R fcvt.s.d
0100001		00000	rs1	rm	rd	1010011	R fcvt.d.s
1010001		rs2	rs1	010	rd	1010011	R feq.d
1010001		rs2	rs1	001	rd	1010011	R flt.d
1010001		rs2	rs1	000	rd	1010011	R fle.d
1110001		00000	rs1	001	rd	1010011	R fclass.d
1100001		00000	rs1	rm	rd	1010011	R fcvt.w.d
1100001		00001	rs1	rm	rd	1010011	R fcvt.wu.d
1101001		00000	rs1	rm	rd	1010011	R fcvt.d.w
1101001		00001	rs1	rm	rd	1010011	R fcvt.d.wu

รูปที่ 5.12: แผนผังรหัสดำเนินการ ของ RV32D มีโครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง มีคำสั่งบางคำสั่งในสองรูปนี้ ที่ไม่แตกต่างกันตามความกว้างของข้อมูล รูปนี้มีคำสั่ง `fcvt.s.d` และ `fcvt.d.s` เฉพาะตัว ในขณะที่อีกรูปมีคำสั่ง `fmv.xw` และ `fmv.wx` (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I flh
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsh
rs3	10	rs2	rs1	rm	rd	1000011	R4 fmadd.h
rs3	10	rs2	rs1	rm	rd	1000111	R4 fmsub.h
rs3	10	rs2	rs1	rm	rd	1001011	R4 fnmsub.h
rs3	10	rs2	rs1	rm	rd	1001111	R4 fnmadd.h
0000010		rs2	rs1	rm	rd	1010011	R fadd.h
0000110		rs2	rs1	rm	rd	1010011	R fsub.h
0001010		rs2	rs1	rm	rd	1010011	R fmul.h
0001110		rs2	rs1	rm	rd	1010011	R fdiv.h
0101110		00000	rs1	rm	rd	1010011	R fsqrt.h
0010010		rs2	rs1	000	rd	1010011	R fsgnj.h
0010010		rs2	rs1	001	rd	1010011	R fsgnjn.h
0010010		rs2	rs1	010	rd	1010011	R fsgnjx.h
0010110		rs2	rs1	000	rd	1010011	R fmin.h
0010110		rs2	rs1	001	rd	1010011	R fmax.h
0100000		00010	rs1	rm	rd	1010011	R fcvt.s.h
0100010		00000	rs1	rm	rd	1010011	R fcvt.h.s
0100001		00010	rs1	rm	rd	1010011	R fcvt.d.h
0100010		00001	rs1	rm	rd	1010011	R fcvt.h.d
1010010		rs2	rs1	010	rd	1010011	R feq.h
1010010		rs2	rs1	001	rd	1010011	R flt.h
1010010		rs2	rs1	000	rd	1010011	R fle.h
1110010		00000	rs1	001	rd	1010011	R fclass.h
1100010		00000	rs1	rm	rd	1010011	R fcvt.w.h
1100010		00001	rs1	rm	rd	1010011	R fcvt.wu.h
1101010		00000	rs1	rm	rd	1010011	R fcvt.h.w
1101010		00001	rs1	rm	rd	1010011	R fcvt.h.wu
1110010		00000	rs1	000	rd	1010011	R fmv.x.h
1111010		00000	rs1	000	rd	1010011	R fmv.h.x

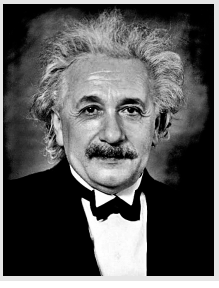
รูปที่ 5.13: แผนผังรหัสดำเนินการ Zfh มีรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ และชื่อคำสั่ง รูปนี้รวมถึงคำสั่งทั้งสองประเภทที่พบเฉพาะในรูปก่อนหน้า (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มารูปนี้)

6 RV32A: คำสั่งอะตอมิก

ทุกสิ่งควรทำให้เรียบง่ายที่สุดเท่าที่เป็นไปได้ แต่ไม่เรียบง่ายเกินไป

—Albert Einstein, 1933

อัลเบิร์ต ไอส์สไตลน์ (1879–1955) เป็น นักวิทยาศาสตร์ ที่ มีชื่อ เสียงที่สุดในศตวรรษที่ 20 เขาคิดค้นทฤษฎีสัมพัทธภาพและสนับสนุนการสร้างระเบิดปรมาณู ในสงครามโลกครั้งที่สอง



6.1 บทนำ

เราถือว่าผู้อ่านเข้าใจเกี่ยวกับการสนับสนุนสถาปัตยกรรมชุดคำสั่งสำหรับการประมวลผลแบบหลายตัวอยู่แล้ว ดังนั้นในบทนี้จะอธิบายคำสั่ง RV32A และการทำงานของคำสั่งอะตอมิก หากผู้อ่านรู้สึกว่ายังไม่มีความรู้พื้นฐานเพียงพอหรือต้องการทบทวน ให้ศึกษาหัวข้อ “การซิงโครไนซ์ (วิทยาการคอมพิวเตอร์)” บนวิกิพีเดีย (https://en.wikipedia.org/wiki/Synchronization_(computer_science)) หรืออ่านหัวข้อที่ 2.1 ของหนังสือสถาปัตยกรรม RISC-V ที่เกี่ยวข้องของเรา [Patterson and Hennessy 2021]

RV32A มีคำสั่งอะตอมิก 2 ประเภท สำหรับการซิงโครไนซ์:

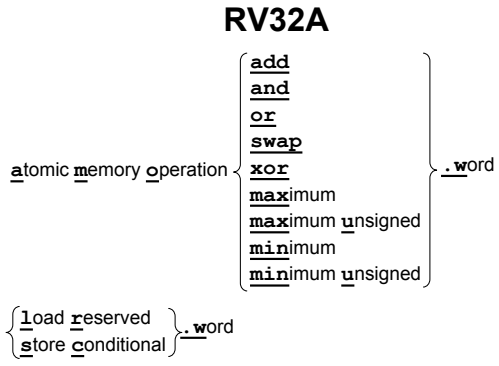
- การดำเนินการหน่วยความจำแบบอะตอมิก (Atomic Memory Operations - AMO) และ
- การโหลดแบบจอง (Load Reserved) และการบันทึกแบบมีเงื่อนไข (Store Conditional)

รูป 6.1 เป็นการแสดงภาพของชุดคำสั่งเสริม RV32A และรูป 6.2 แสดงรายการรหัสดำเนินการ (opcodes) และรูปแบบคำสั่ง

คำสั่ง AMO จะดำเนินการกับตัวถูกดำเนินการในหน่วยความจำแบบอะตอมิก และตั้งค่ารีจิสเตอร์ปลายทางเป็นคำสั่งดั้งเดิมของหน่วยความจำ คำว่า อะตอมิก หมายถึงไม่มีการขัดจังหวะระหว่างการอ่านและการเขียนหน่วยความจำ และโปรเซสเซอร์อื่นๆ ไม่สามารถแก้ไขค่าหน่วยความจำ ระหว่างการอ่านและการเขียนของคำสั่ง AMO ได้

ทั้ง AMO และ LR/ SC ต้องการตำแหน่งที่อยู่หน่วยความจำที่จัดแนวตามธรรมชาติ เนื่องจากการรับประกันความเป็นอะตอมิก (atomicity) ข้ามขอบเขตของแคชไลน์ (cache-line boundaries) เป็นเรื่องที่ยุ่งยากสำหรับฮาร์ดแวร์

การโหลดแบบจองและการบันทึกแบบมีเงื่อนไข ดำเนินการแบบอะตอมิกผ่านสองคำสั่ง: การโหลดแบบจอง (Load reserved) อ่านเวิร์ดจากหน่วยความจำ เขียนลงในรีจิสเตอร์ปลายทาง และบันทึกการจอง (reservation) สำหรับคำนั้นในหน่วยความจำ การบันทึกแบบมีเงื่อนไข (Store conditional) บันทึกเวิร์ดที่ตำแหน่งที่อยู่ในรีจิสเตอร์ต้นทาง



รูปที่ 6.1: ไดอะแกรมของคำสั่ง RV32A

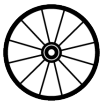
31 27	26	25	24 20	19 15	14 12	11 7	6 0	
00010	aq	rl	00000	rs1	010	rd	0101111	R lr.w
00011	aq	rl	rs2	rs1	010	rd	0101111	R sc.w
00001	aq	rl	rs2	rs1	010	rd	0101111	R amoswap.w
00000	aq	rl	rs2	rs1	010	rd	0101111	R amoadd.w
00100	aq	rl	rs2	rs1	010	rd	0101111	R amoxor.w
01100	aq	rl	rs2	rs1	010	rd	0101111	R amoand.w
01000	aq	rl	rs2	rs1	010	rd	0101111	R amoor.w
10000	aq	rl	rs2	rs1	010	rd	0101111	R amomin.w
10100	aq	rl	rs2	rs1	010	rd	0101111	R amomax.w
11000	aq	rl	rs2	rs1	010	rd	0101111	R amominu.w
11100	aq	rl	rs2	rs1	010	rd	0101111	R amomaxu.w

รูปที่ 6.2: แผนผังรหัสดำเนินการ RV32A มีโครงสร้างรหัสดำเนินการ, รูปแบบคำสั่ง ,ประเภทรูปแบบ และชื่อ (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

โดยจะทำได้ก็ต่อเมื่อมีการจองการโหลดสำหรับที่อยู่นั้นอยู่ ถ้าการบันทึกสำเร็จจะเขียนศูนย์ลงในรีจิสเตอร์ปลายทาง หรือจะเขียนรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ ถ้าการบันทึกไม่สำเร็จ

คำถาม: ทำไม RV32A มีสองวิธีในการดำเนินการแบบอะตอมิก คำตอบคือมีกรณีการใช้งานที่แตกต่างกันสองแบบ

นักพัฒนาภาษาการเขียนโปรแกรมสมมติว่าโครงสร้างพื้นฐานสามารถดำเนินการเปรียบเทียบและสลับแบบอะตอมิก (atomic compare-and-swap) ได้:: เปรียบเทียบค่าจากรีจิสเตอร์กับค่าในหน่วยความจำที่ระบุโดยรีจิสเตอร์อื่น และหากเท่ากัน ให้สลับค่าจากรีจิสเตอร์ที่สามกับค่าในหน่วยความจำ การสมมตินี้เกิดจากความเป็นสากลของ compare-and-swap ซึ่งเป็นตัวเลือกพื้นฐานสำหรับการซิงโครไนซ์ เพราะสามารถสังเคราะห์การดำเนินการซิงโครไนซ์เวิร์ดเดียวได้ [Herlihy 1991]


ความเรียบง่าย

อย่างไรก็ตาม การเพิ่มคำสั่ง compare investigaciones System: compare-and-swap ต้องใช้รีจิสเตอร์ต้นทางสามตัวในคำสั่งเดียว ซึ่งทำให้การออกแบบส่วนต่อประสานหน่วยความจำ ทางเดินข้อมูลจำนวนเต็ม และรูปแบบคำสั่งซับซ้อนขึ้น (คำสั่ง multiply-add ของ RV32FD ที่ใช้ตัวถูกดำเนินการสามตัวส่งผลต่อทางเดินข้อมูลทศนิยม ไม่ใช่ทางเดินข้อมูลจำนวนเต็ม) โชคดีที่ Load Reserved และ Store Conditional ใช้รีจิสเตอร์ต้นทางเพียงสองตัวและสามารถนำมาใช้เพื่อสร้าง compare-and-swap ได้ (ดูครึ่งบนของรูปที่ 6.3)


ประสิทธิภาพ

เหตุผลที่ RV32A มีคำสั่ง AMO ด้วยก็คือ คำสั่งเหล่านี้สามารถปรับขนาดได้ดีกว่าคำสั่ง LR/SC (Load Reserved และ Store Conditional) ในระบบมัลติโปรเซสเซอร์ขนาดใหญ่เมื่อเทียบกับการโหลดแบบสงวนไว้และการบันทึกแบบมีเงื่อนไข นอกจากนี้คำสั่ง AMO ยังสามารถใช้เพื่อดำเนินการลดขนาด (reduction operations) ได้อย่างมีประสิทธิภาพ และยังมีประโยชน์ในการสื่อสารกับอุปกรณ์ I/O เนื่องจากมันทำการอ่านและเขียนในธุรกรรมบัสแบบอะตอมิกครั้งเดียว ความเป็นอะตอมิกนี้ช่วยลดความซับซ้อนของไดรเวอร์อุปกรณ์และเพิ่มประสิทธิภาพ I/O ส่วนล่างของรูปที่ 6.3 แสดงวิธีเขียนส่วนวิกฤต (critical section) โดยใช้การสลับแบบอะตอมิก

■ ***รายละเอียดเพิ่มเติม: โมเดลความสอดคล้องของหน่วยความจำ***

RISC-V ใช้แบบจำลองความสอดคล้องของหน่วยความจำแบบผ่อนปรน (relaxed memory consistency model) ดังนั้นเธรดอื่นอาจเห็นการเข้าถึงหน่วยความจำบางส่วนในลำดับที่ไม่ตรงกัน รูปที่ 6.2 แสดงว่าคำสั่ง RV32A ทั้งหมดมี (`aq`) *บิตการเข้าถึง - acquire bit* และ (`rl`) *บิตการปล่อย - release bit* การดำเนินการอะตอมที่มีการตั้งบิต `aq` จะรับประกันว่าเธรดอื่นจะเห็น AMO ตามลำดับการเข้าถึงหน่วยความจำ *ที่ตามมา* ถ้าตั้งบิต `rl` เธรดอื่นจะเห็นการดำเนินการของอะตอมตามตามลำดับกับการเข้าถึงหน่วยความจำ *ก่อนหน้า* หากต้องการเรียนรู้เพิ่มเติม [Adve and Gharachorloo 1996] เป็นบทเรียนที่ดีสำหรับหัวข้อนี้

อะไรที่แตกต่าง? MIPS-32 เดิมไม่มีกลไกสำหรับการซิงโครไนซ์ แต่สถาปนิกได้เพิ่มคำสั่ง โหลดแบบจอง/บันทึกแบบมีเงื่อนไข ใน MIPS รุ่นหลัง

```
# Compare-and-swap (CAS) memory word M[a0] using lr/sc instructions
# Expected old value in a1 desired new value in a2
   0: 100526af     lr.w  a3,(a0)      # โหลดค่าเก่า
   4: 06b69e63     bne   a3,a1,80     # เปรียบเทียบค่าเก่ากับ a1
   8: 18c526af     sc.w  a3,a2,(a0)   # ถ้าเท่ากันให้เก็บค่าใหม่
   c: fe069ae3     bnez  a3,0         # ถ้าไม่สำเร็จให้ลองใหม่
        … เปรียบเทียบ - รหัสหลังจากสลับสำเร็จ …
  80:                                 # เปรียบเทียบ-สลับไม่สำเร็จ


# Critical section guarded by test-and-set spinlock using an คำสั่ง AMO
   0: 00100293     li            t0,1         # เริ่มต้นการล็อค
   4: 0c55232f     amoswap.w.aq t1,t0,(a0)    # ลองรับล็อคเกอร์
   8: fe031ee3     bnez          t1,4         # ถ้าพลาดก็พยายามต่อไป
        … รหัสส่วนสำคัญ …
  20: 0a05202f     amoswap.w.rl x0,x0,(a0)    # ปลดล็อค
```

รูปที่ 6.3: ตัวอย่างการซิงโครไนซ์สองแบบ ตัวอย่างแรกใช้ การโหลดแบบสงวนและการบันทึกแบบมีเงื่อนไข (**load reserved/store conditional**) `lr.w`, `sc.w` เพื่อสร้างการเปรียบเทียบและสลับ (compare-and-swap) และตัวอย่างที่สองใช้การสลับแบบอะตอมิก `amoswap.w` เพื่อสร้าง mutex

6.2 ข้อสังเกต สรุป

RV32A เป็นชุดคำสั่งเสริมที่ไม่บังคับ และโปรเซสเซอร์ RISC-V จะง่ายขึ้นถ้าไม่มี RV32A อย่างไรก็ตาม ตามที่ไอน์สไตน์กล่าว ทุกอย่างควรเรียบง่าย *เท่าที่เป็นไปได้* แต่ไม่เรียบง่ายเกินไป สถานการณ์หลายอย่างจำเป็นต้องใช้ RV32A

6.3 เรียนรู้เพิ่มเติม

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66--76, 1996.

M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

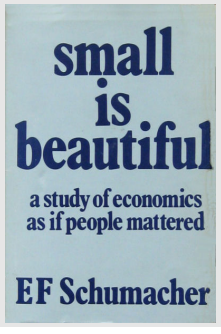
A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

7 RV32C: คำสั่งแบบบีบอัด

เล็กๆ ก็สวยงาม

—E.F. ชูมัคเกอร์, 2520

E.F. ชูมัคเกอร์ (พ.ศ. 2454–2520) เขียนหนังสือเศรษฐศาสตร์เล่มนี้ที่สนับสนุนเทคโนโลยีที่เหมาะสมกับขนาดมนุษย์ การกระจายอำนาจและเทคโนโลยีที่เหมาะสม หนังสือนี้ได้รับการแปลเป็นภาษาต่างๆ หลายภาษา และได้รับการยกย่องให้เป็น 1 ใน 100 หนังสือที่มีอิทธิพลมากที่สุดนับตั้งแต่สงครามโลกครั้งที่สอง





ขนาดโปรแกรม



ความเรียบง่าย

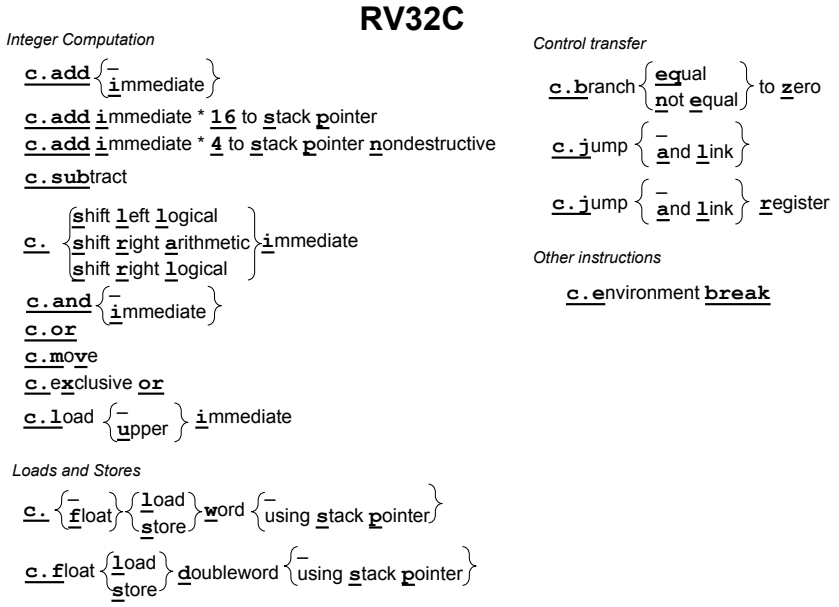
7.1 บทนำ

สถาปัตยกรรมชุดคำสั่งก่อนหน้านี้ ได้ขยายจำนวนคำสั่ง และรูปแบบคำสั่งอย่างมากเพื่อลดขนาดโค้ด เช่น การเพิ่มคำสั่งสั้นที่มีตัวถูกดำเนินการ 2 ตัว แทนที่จะมี 3 ตัว, ฟิลด์ค่าคงทันทีขนาดเล็ก และอื่นๆ ARM และ MIPS ได้พัฒนา ISA ใหม่ถึงสองครั้ง เพื่อลดขนาดโค้ด: ARM Thumb และ Thumb-2 รวมถึง MIPS16 และ microMIPS สถาปัตยกรรมชุดคำสั่งเหล่านี้ทำให้โปรเซสเซอร์และคอมไพเลอร์ซับซ้อนขึ้น และเพิ่มภาระด้านความเข้าใจให้กับโปรแกรมเมอร์ภาษาแอสเซมบลี

RV32C ใช้แนวทางใหม่: *ทุก*คำสั่งสั้น 16 บิต *ต้อง*แมปไปยังคำสั่ง RISC-V มาตรฐาน 32 บิต *เพียงคำสั่งเดียว* โดยแอสเซมเบลอร์และลิงเกอร์เท่านั้นที่รับรู้คำสั่ง 16 บิต และขึ้นอยู่กับมันที่จะแทนคำสั่ง RISC-V 32 บิตด้วยคำสั่งคำสั่ง 16 บิตที่เทียบเท่า นักเขียนคอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลีสามารถไม่รู้ถึงคำสั่ง RV32C และรูปแบบของมันได้เลย แต่รู้ว่าผลลัพธ์คือโปรแกรมที่มีขนาดเล็กกว่าส่วนใหญ่ รูปที่ 7.1 เป็นการแสดงภาพของชุดคำสั่งส่วนขยาย RV32C

สถาปนิก RISC-V เลือกคำสั่งในส่วนขยาย RVC เพื่อให้ได้การบีบอัดโค้ดที่ดีใน โปรแกรมหลากหลาย โดยใช้ข้อสังเกตสามประการเพื่อให้คำสั่งอยู่ใน 16 บิต: ประการแรก รีจิสเตอร์ที่นิยมสิบตัว (`a0–a5`, `s0–s1`, `sp` และ `ra`) ถูกเข้าถึงบ่อยกว่ารีจิสเตอร์อื่นๆ ประการที่สอง คำสั่งหลายคำสั่งเขียนทับตัวถูกดำเนินการต้นทางตัวหนึ่ง ประการที่สาม ตัวเลขค่าคงทันที (immediate) มักมีขนาดเล็ก และบางคำสั่งชอบค่าคงที่ทันทีเฉพาะค่า ดังนั้น คำสั่ง RV32C หลายคำสั่งสามารถเข้าถึงได้เฉพาะรีจิสเตอร์ที่นิยม บางคำสั่งเขียนทับตัวถูกดำเนินการต้นทางโดยปริยาย และค่าคงที่ทันทีเกือบทั้งหมดมีขนาดเล็กลง โดยการโหลดและบันทึกใช้เฉพาะออฟเซ็ตที่เป็นจำนวนเต็มบวกในหน่วยที่เป็นทวีคูณของขนาดตัวถูกดำเนินการ

รูปที่ 7.3 และ 7.4 ตัวอย่างโค้ดที่คอมไพล์สำหรับ Insertion Sort และ DAXPY โดยใช้ RV32C เราแสดงคำสั่ง RV32C เพื่อแสดงผลกระทบของการบีบอัดอย่างชัดเจน แต่ปกติคำ



รูปที่ 7.1: แผนภาพของคำสั่ง RV32C ฟิลด์ค่าคงที่ทันทีของคำสั่งเลื่อนบิต `c.addi4spn` ถูกขยายศูนย์และถูกขยายเครื่องหมายสำหรับคำสั่งอื่นๆ

สั่งเหล่านี้จะมองไม่เห็นในโปรแกรมภาษาแอสเซมบลี ความคิดเห็นจะแสดงคำสั่ง 32 บิตที่เทียบเท่ากับคำสั่ง RV32C ในวงเล็บ ภาคผนวก A รวมถึงคำสั่ง RISC-V แบบ 32 บิตที่สอดคล้องกับคำสั่ง RV32C 16 บิตแต่ละคำสั่ง

ตัวอย่างเช่น ที่ตำแหน่งที่อยู่ 4 ใน Insertion Sort ในรูป 7.3 แอสเซมเบลอร์แทนที่คำสั่ง RV32I 32 บิตนี้:

```
addi a4,x0,1  # i = 1
```

ด้วยคำสั่ง RV32C 16 บิตนี้:

```
c.li a4,1     # (ขยายเป็น addi a4,x0,1) i = 1
```

คำสั่งโหลดค่าคงที่ทันที ของ RV32C มีขนาดแคบลง เพราะต้องการระบุเพียงรีจิสเตอร์เดียวและค่าคงที่ทันทีขนาดเล็ก รหัสเครื่องของคำสั่ง `c.li` มีเพียงสี่หลักแบบเลขฐานสิบหก ในรูป 7.3 แสดงว่าคำสั่งว่า `c.li` มีความยาวเพียง 2 ไบต์

อีกตัวอย่างคือ ที่ตำแหน่งที่อยู่ 10 ในรูป 7.3 ซึ่งแอสเซมเบลอร์แทนที่:

```
add a2,x0,a3  # a2 เป็นตัวชี้ไปที่ a[j]
```

ด้วยคำสั่ง RV32C 16 บิตนี้:

```
c.mv a2,a3    # (ขยายเป็น add a2,x0,a3) a2 เป็นตัวชี้ไปที่ a[j]
```

คำสั่ง move ของ RV32C มีความยาวเพียง 16 บิต เพราะระบุเพียงสองรีจิสเตอร์

Benchmark	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Insertion Sort	Instructions	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instructions	10	12	16	11
	Bytes	28	32	50	28

รูปที่ 7.2: คำสั่งและขนาดของโค้ดสำหรับ Insertion Sort และ DAXPY ใน ISA แบบบีบอัด



ในขณะที่นักออกแบบโปรเซสเซอร์ไม่สามารถละเลยคำสั่ง RV32C ได้ เทคนิคหนึ่งที่ทำให้การใช้งานมีต้นทุนต่ำ: ตัวถอดรหัส (decoder) จะแปลคำสั่ง 16 บิตทั้งหมดเป็นคำสั่ง 32 บิตที่เทียบเท่า *ก่อน*การดำเนินการ รูป 7.6 ถึง 7.8 ระบุรูปแบบคำสั่ง RV32C และรหัสดำเนินการที่ตัวถอดรหัสแปล ซึ่งเทียบเท่ากับเกตเพียง 400 ตัว เมื่อโปรเซสเซอร์ 32 บิตที่เล็กที่สุด—โดยไม่มีส่วนขยาย RISC-V—มีเกต 8000 ตัว หากคิดเป็น 5% ของการออกแบบขนาดเล็กเช่นนี้ ตัวถอดรหัสแทบจะหายไปในโปรเซสเซอร์ขนาดปานกลางที่มีแคช ซึ่งมีเกตประมาณ 100,000 เกต

อะไรที่แตกต่าง ไม่มีคำสั่งแบบไบต์หรือครึ่งเวิร์ด ใน RV32C เพราะคำสั่งอื่นมีผลกระทบต่อขนาดโค้ดมากกว่า ข้อได้เปรียบด้านขนาดเล็กของ Thumb-2 เหนือ RV32C ในรูป 1.5 หน้าที่ 11 มาจากการประหยัดขนาดโค้ดของคำสั่งโหลดและบันทึกหลายรายการ **Load and Store Multiple** เมื่อเข้าและออกจากโพรซีเดอร์ RV32C ไม่รวมคำสั่งเหล่านี้เพื่อรักษาการแมปแบบหนึ่งต่อหนึ่งกับคำสั่ง RV32G ซึ่งละเว้นคำสั่งเหล่านี้เพื่อลดความซับซ้อนในการใช้งานสำหรับโปรเซสเซอร์ระดับสูง เนื่องจาก Thumb-2 เป็น ISA แยกจาก ARM-32 แต่โปรเซสเซอร์สามารถสลับระหว่างทั้งสองได้ ฮาร์ดแวร์จึงต้องมีตัวถอดรหัสคำสั่งสองตัว: หนึ่งสำหรับ ARM-32 และอีกหนึ่งสำหรับ Thumb-2 แต่ RV32GC เป็น ISA เดียว ดังนั้นโปรเซสเซอร์ RISC-V ต้องการตัวถอดรหัสเพียงตัวเดียว

■ ***รายละเอียดเพิ่มเติม: ทำไมสถาปนิกถึงข้าม RV32C?***

การถอดรหัสคำสั่งอาจเป็นคอขวดสำหรับโปรเซสเซอร์ซูเปอร์สเกลาร์ ที่พยายามดึงคำสั่งหลายคำสั่งต่อรอบสัญญาณนาฬิกา ตัวอย่างเพิ่มเติมคือ มาโครฟิวชัน *macrofusion* ซึ่งตัวถอดรหัสคำสั่งรวมคำสั่ง RISC-V เข้าด้วยกันเพื่อสร้างคำสั่งที่มีประสิทธิภาพมากขึ้นสำหรับการดำเนินการ (ดูบท 1) การผสมผสานระหว่างคำสั่ง RV32C 16 บิต และคำสั่ง RV32I 32 บิตอาจทำให้การถอดรหัสที่ซับซ้อนยากต่อการเสร็จสิ้น ภายในรอบสัญญาณนาฬิกาของการใช้งานประสิทธิภาพสูง

7.2 การเปรียบเทียบ RV32GC, Thumb-2, microMIPS และ x86-32

รูปที่ 7.2 ตารางสรุปขนาดของ Insertion Sort และ DAXPY สำหรับ ISA ทั้ง 4 แบบ

จากคำสั่ง RV32I เดิม 19 คำสั่ง ใน Insertion Sort มี 12 คำสั่ง ที่กลายเป็น RV32C ทำให้ขนาดโค้ดลดลงจาก $19 \times 4 = 76$ ไบต์เหลือ $12 \times 2 + 7 \times 4 = 52$ ไบต์ ประหยัดได้ $24/76 = 32\%$. สำหรับ DAXPY ขนาดโค้ดลดลงจาก $11 \times 4 = 44$ ไบต์เหลือ $8 \times 2 + 3 \times 4 = 28$ ไบต์ หรือประหยัดได้ $16/44 = 36\%$

ผลลัพธ์สำหรับตัวอย่างเล็กๆ สองตัวนี้ สอดคล้องอย่างน่าประหลาดใจกับรูป 1.5 ในหน้า 11 บทที่ 1 ซึ่งแสดงว่าโค้ด RV32G มีขนาดใหญ่กว่าโค้ด RV32GC ประมาณ 37% สำหรับชุดโปรแกรมที่ใหญ่กว่ามาก เพื่อให้ได้การประหยัดในระดับนั้น มากกว่าครึ่งหนึ่งของคำสั่งในโปรแกรมต้องเป็นคำสั่ง RV32C

■ ***รายละเอียดเพิ่มเติม: RV32C มีความเป็นเอกลักษณ์จริงหรือไม่?***

คำสั่ง RV32I จะแยกไม่ออกใน RV32IC Thumb-2 เป็น ISA แยกต่างหากที่มีคำสั่ง 16 บิต รวมถึงส่วนใหญ่ (แต่ไม่ทั้งหมด) ของ ARMv7 ตัวอย่างเช่น คำสั่ง*Compare and Branch on Zero* มีใน Thumb-2 แต่ไม่มีใน ARMv7 ในขณะที่คำสั่ง Reverse Subtract with Carry มีใน ARMv7 แต่ไม่มีใน Thumb-2 เช่นกัน microMIPS32 ก็ไม่ใช่ซูเปอร์เซ็ตของ MIPS32 ตัวอย่างเช่น microMIPS คูณระยะการกระโดดด้วย 2 แต่ใน MIPS32 จะคูณด้วย 4 ส่วน RISC-V จะคูณด้วย 2 เสมอ

7.3 ข้อสังเกตสรุป

ฉันจะเขียนจดหมายที่สั้นกว่านี้ แต่ฉันไม่มีเวลา

—แบลส ปาสกาล, 1656.



ขนาดโปรแกรม

เขาเป็นนักคณิตศาสตร์ที่สร้างเครื่องคิดเลขเชิงกลเครื่องแรก ซึ่งนำไปสู่การตั้งชื่อภาษาการเขียนโปรแกรมโดย Niklaus Wirth ผู้ได้รับรางวัลทัวริง

RV32C ทำให้ RISC-V มีขนาดโค้ดที่เล็กที่สุดในปัจจุบัน เกือบจะถือได้ว่า RV32C เป็นคำสั่งเทียมที่ได้รับการช่วยเหลือจากฮาร์ดแวร์ อย่างไรก็ตาม ตอนนี้แอสเซมเบลอร์ซ่อนคำสั่งเหล่านี้จากโปรแกรมเมอร์ภาษาแอสเซมบลีและนักเขียนคอมไพเลอร์ แทนที่จะขยายชุดคำสั่งจริงด้วยการดำเนินการยอดนิยมที่ทำให้โค้ด RISC-V ใช้งานและอ่านง่ายขึ้น เหมือนในบท 3 ทั้งสองแนวทางนี้ช่วยเพิ่มประสิทธิภาพการทำงานของโปรแกรมเมอร์



สง่างาม

เราถือว่า RV32C เป็นหนึ่งในตัวอย่างที่ดีที่สุดของ RISC-V ที่มีกลไกที่เรียบง่ายแต่ทรงพลัง ซึ่งช่วยปรับปรุงประสิทธิภาพด้านต้นทุน

7.4 เรียนรู้เพิ่มเติม

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

```
# RV32C (19 คำสั่ง, 52 bytes)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
  0: 00450693 addi   a3,a0,4   # a3 เป็นตัวชี้ไปที่ a[i]
  4: 4705     c.li   a4,1      # (ขยายเป็น addi a4,x0,1) i = 1
Outer Loop:
  6: 00b76363 bltu   a4,a1,c   # ถ้า i < n กระโดดไปที่ Continue Outer loop
  a: 8082     c.ret            # (ขยายเป็น jalr x0,ra,0) ออกจากฟังก์ชัน
Continue Outer Loop:
  c: 0006a803 lw     a6,0(a3)  # x = a[i]
 10: 8636     c.mv   a2,a3     # (ขยายเป็น add a2,x0,a3) a2 เป็นตัวชี้ไปที่ a[j]
 12: 87ba     c.mv   a5,a4     # (ขยายเป็น add a5,x0,a4) j = i
InnerLoop:
 14: ffc62883 lw     a7,-4(a2) # a7 = a[j-1]
 18: 01185763 ble    a7,a6,26  # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit InnerLoop
 1c: 01162023 sw     a7,0(a2)  # a[j] = a[j-1]
 20: 17fd     c.addi a5,-1     # (ขยายเป็น addi a5,a5,-1) j--
 22: 1671     c.addi a2,-4     # (ขยายเป็น addi a2,a2,-4) ลด a2 ชี้ไปที่ a[j]
 24: fbe5     c.bnez a5,14     # (ขยายเป็น bne a5,x0,14) ถ้า j != 0 กระโดดไปที่ InnerLoop
Exit InnerLoop:
 26: 078a     c.slli a5,0x2    # (ขยายเป็น slli a5,a5,0x2) คูณ a5 by 4
 28: 97aa     c.add  a5,a0     # (ขยายเป็น add a5,a5,a0) a5 = byte address of a[j]
 2a: 0107a023 sw     a6,0(a5)  # a[j] = x
 2e: 0705     c.addi a4,1      # (ขยายเป็น addi a4,a4,1) i++
 30: 0691     c.addi a3,4      # (ขยายเป็น addi a3,a3,4) เพิ่ม a3 ชี้ไปที่ a[i]
 32: bfd1     c.j    6         # (ขยายเป็น jal x0,6) กระโดดไปที่ Outer Loop
```

รูปที่ 7.3: รหัส RV32C สำหรับการเรียงลำดับการแทรก ที่รูป 2.5 คำสั่ง 16 บิต 12 คำสั่ง ทำให้โค้ดเล็กลง 32% ขนาดของแต่ละคำสั่งเห็นได้จากจำนวนอักขระฐานสิบหกในคอลัมน์ที่สอง คำสั่ง RV32C (เริ่มต้นด้วย `c.`) จะแสดงอย่างชัดเจนในตัวอย่างนี้ แต่โดยปกติโปรแกรมเมอร์และคอมไพเลอร์ภาษาแอสเซมบลีจะมองไม่เห็น

```
# RV32DC (11 คำสั่ง, 28 bytes)
# a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], fa0 คือ a
  0: cd09     c.beqz a0,1a        # (ขยายเป็น beq a0,x0,1a) ถ้า n==0, กระโดดไปที่ Exit
  2: 050e     c.slli a0,a0,0x3    # (ขยายเป็น slli a0,a0,0x3) a0 = n*8
  4: 9532     c.add a0,a2         # (ขยายเป็น add a0,a0,a2) a0 = ที่อยู่ของ y[n]
Loop:
  6: 2218     c.fld fa4,0(a2)     # (ขยายเป็น fld fa4,0(a2)) fa5 = x[]
  8: 219c     c.fld fa5,0(a1)     # (ขยายเป็น fld fa5,0(a1)) fa4 = y[]
  a: 0621     c.addi a2,8         # (ขยายเป็น addi a2,a2,8) a2++ (เพิ่มตัวชี้ไปที่ y)
  c: 05a1     c.addi a1,8         # (ขยายเป็น addi a1,a1,8) a1++ (เพิ่มตัวชี้ไปที่ x)
  e: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 12: fef63c27 fsd fa5,-8(a2)      # y[i] = a*x[i] + y[i]
 16: fea618e3 bne a2,a0,6         # ถ้า i != n กระโดดไปที่ Loop
Exit:
 1a: 8082     ret                 # (ขยายเป็น jalr x0,ra,0) ออกจากฟังก์ชัน
```

รูปที่ 7.4: รหัส RV32DC สำหรับ DAXPY ที่รูป 2.5 8 คำสั่ง ขนาด 16 บิต ทำให้โค้ดเล็กลง 36% ขนาดของแต่ละคำสั่งนั้นเห็นได้จากจำนวนอักขระฐานสิบหกในคอลัมน์ที่สอง คำสั่ง RV32C (เริ่มต้นด้วย `c.`) แสดงอย่างชัดเจนในตัวอย่างนี้ แต่โดยปกติโปรแกรมเมอร์และคอมไพเลอร์ภาษาแอสเซมบลีจะมองไม่เห็น

15–13	12	11–7	6–5	4–2	1–0	
000	nzimm[5]	0		nzimm[4:0]	01	CI c.nop
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]	01	CI c.addi
001	imm[11|4|9:8|10|6|7|3:1|5]				01	CJ c.jal
010	imm[5]	rd≠0		imm[4:0]	01	CI c.li
011	nzimm[9]	2		nzimm[4|6|8:7|5]	01	CI c.addi16sp
011	nzimm[17]	rd≠{0,2}		nzimm[16:12]	01	CI c.lui
100	nzuimm[5]	00 rs1′/rd′		nzuimm[4:0]	01	CI c.srli
100	nzuimm[5]	01 rs1′/rd′		nzuimm[4:0]	01	CI c.srai
100	imm[5]	10 rs1′/rd′		imm[4:0]	01	CI c.andi
100	0	11 rs1′/rd′	00	rs2′	01	CR c.sub
100	0	11 rs1′/rd′	01	rs2′	01	CR c.xor
100	0	11 rs1′/rd′	10	rs2′	01	CR c.or
100	0	11 rs1′/rd′	11	rs2′	01	CR c.and
101	imm[11|4|9:8|10|6|7|3:1|5]				01	CJ c.j
110	imm[8|4:3]	rs1′		imm[7:6|2:1|5]	01	CB c.beqz
111	imm[8|4:3]	rs1′		imm[7:6|2:1|5]	01	CB c.bnez

รูปที่ 7.5: แผนผังรหัสดำเนินการ RV32C (**bits**[1 : 0] = 01) แสดงโครงสร้าง opcodes รูปแบบและชื่อ `rd'`, `rs1'` และ `rs2'` หมายถึง 10 รีจิสเตอร์ยอดนิยม `a0–a5`, `s0–s1`, `sp`, และ `ra` (ตารางที่ 12.5 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

15–13	12–10	9–7	6–5	4–2	1–0	
000	0			0	00	CIW *Illegal instruction*
000	nzuimm[5:4|9:6|2|3]			rd′	00	CIW c.addi4spn
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	CL c.fld
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	CL c.lw
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	CL c.flw
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	CL c.fsd
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	CL c.sw
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	CL c.fsw

รูปที่ 7.6: แผนผังรหัสดำเนินการ RV32C (**bits**[1 : 0] = 00) แสดงโครงสร้าง opcodes รูปแบบและชื่อ `rd'`, `rs1'` และ `rs2'` อ้างถึง 10 รีจิสเตอร์ยอดนิยม `a0–a5`, `s0–s1`, `sp` และ `ra` (ตารางที่ 12.4 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

15 13	12	11 7	6 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	CI c.slli
000	0	rs1/rd≠0	0	10	CI c.slli64
001	uimm[5]	rd	uimm[4:3|8:6]	10	CSS c.fldsp
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	CSS c.lwsp
011	uimm[5]	rd	uimm[4:2|7:6]	10	CSS c.flwsp
100	0	rs1≠0	0	10	CJ c.jr
100	0	rd≠0	rs2≠0	10	CR c.mv
100	1	0	0	10	CI c.ebreak
100	1	rs1≠0	0	10	CJ c.jalr
100	1	rs1/rd≠0	rs2≠0	10	CR c.add
101	uimm[5:3|8:6]		rs2	10	CSS c.fsdsp
110	uimm[5:2|7:6]		rs2	10	CSS c.swsp
111	uimm[5:2|7:6]		rs2	10	CSS c.fswsp

รูปที่ 7.7: แผนผังรหัสดำเนินการ RV32C (**bits**[1 : 0] = 10) แสดงรายการโครงสร้าง opcodes รูปแบบและชื่อ (ตารางที่ 12.6 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

Format	Meaning	15 13	12	11 10	9 7	6 5	4 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

รูปที่ 7.8: รูปแบบคำสั่งแบบบีบอัด RVC 16 บิต `rd'`, `rs1'` และ `rs2'` หมายถึง 10 รีจิสเตอร์ยอดนิยม `a0–a5`, `s0–s1`, `sp` และ `ra` (ตารางที่ 12.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

8 RV32V: เวกเตอร์

ฉันสนับสนุนความเรียบง่ายอย่างเต็มที่ ถ้ามันซับซ้อนมาก ฉันก็ไม่เข้าใจ

—ซีมัวร์ เครย์

ซีมัวร์ เครย์ (1925–1996) เป็นสถาปนิกของ Cray-1 ในปี 1976 ซึ่งเป็นซูเปอร์คอมพิวเตอร์ที่ประสบความสำเร็จในเชิงพาณิชย์เครื่องแรก ที่ใช้สถาปัตยกรรมเวกเตอร์ Cray-1 เป็นอัญมณีล้ำค่า และเป็นคอมพิวเตอร์ที่เร็วที่สุดในโลก แม้ว่าจะไม่ได้ใช้คำสั่งเวกเตอร์ก็ตาม





ประสิทธิภาพ

8.1 บทนำ

บทนี้เน้นที่ *ความขนานระดับข้อมูล* (data-level parallelism) ซึ่งมีข้อมูลจำนวนมากที่แอปพลิเคชันสามารถคำนวณได้พร้อมกัน อาร์เรย์เป็นตัวอย่างที่ได้รับความนิยม แม้ว่าจะเป็นพื้นฐานสำหรับแอปพลิเคชันทางวิทยาศาสตร์ แต่โปรแกรมมัลติมีเดียก็ใช้อาร์เรย์เช่นกัน โดยแอปพลิเคชันทางวิทยาศาสตร์มักใช้ข้อมูลจุดทศนิยม ความแม่นยำเท่าเดียวและความแม่นยำสองเท่า ในขณะที่โปรแกรมมัลติมีเดียมักใช้ข้อมูลจำนวนเต็มขนาด 8 บิต และ 16 บิต

สถาปัตยกรรมที่รู้จักกันดีที่สุดสำหรับ ความขนานระดับข้อมูล คือ *คำสั่งเดียว ข้อมูลหลายชุด* (Single Instruction, Multiple Data) SIMD เริ่มได้รับความนิยมครั้งแรก โดยการแบ่งรีจิสเตอร์ขนาด 64 บิต ออกเป็นชิ้นขนาด 8 บิต, 16 บิต หรือ 32 บิตหลายชิ้น แล้วคำนวณบนชิ้นส่วนเหล่านี้แบบขนาน รหัสดำเนินการ (opcode) ระบุความกว้างของข้อมูลและประเภทการดำเนินการ การถ่ายโอนข้อมูลเป็นเพียงการโหลดและบันทึกของเรจิสเตอร์ SIMD เดี่ยวที่กว้าง

ขั้นตอนแรกของการแบ่งรีจิสเตอร์ 64 บิต ที่มีอยู่นั้นน่าดึงดูดเพราะมันตรงไปตรงมา เพื่อทำให้ SIMD เร็วขึ้น สถาปนิกจึงขยายขนาดรีจิสเตอร์ให้กว้างขึ้นเพื่อประมวลผลชิ้นส่วนมากขึ้นพร้อมกัน เนื่องจาก ISA ของ SIMD เป็นส่วนหนึ่งของแนวทางการออกแบบแบบเพิ่มหน่วย (incremental school of design) และ opcode ระบุความกว้างของข้อมูล การขยายเรจิสเตอร์ SIMD ยังทำให้ชุดคำสั่ง SIMD ขยายตามไปด้วย แต่ละขั้นตอนต่อมาของการเพิ่มความกว้างของเรจิสเตอร์ SIMD และจำนวนคำสั่ง SIMD เป็นสองเท่านำ ISA ไปสู่เส้นทางของความซับซ้อนที่เพิ่มขึ้น ซึ่งเป็นภาระของนักออกแบบโปรเซสเซอร์ นักพัฒนาคอมไพเลอร์ และโปรแกรมเมอร์ภาษาแอสเซมบลี

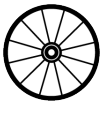
ส่วนขยายมัลติมีเดียของ Intel (MMX) ในปี 1997 ทำให้ SIMD ได้รับความนิยม มันถูกยอมรับและขยายต่อผ่าน Streaming SIMD Extensions (SSE) ในปี 1999 และ Advanced Vector Extensions (AVX) ในปี 2010 ความนิยมของ MMX ได้รับแรงหนุนจากแคมเปญโฆษณาของ Intel ที่แสดงภาพพนักงานสายการผลิตเซมิคอนดักเตอร์เต้นดิสโก้ในชุดสะอาดสีสันสดใส (https://www.youtube.com/watch?v=5zyjSBSvqPc)

ทางเลือกที่เก่ากว่าและในความเห็นของเรา สง่างามมากกว่า ในการใช้ประโยชน์จากความขนานระดับข้อมูล คือสถาปัตยกรรม*เวกเตอร์* บทนี้ให้เหตุผลของเราสำหรับการเลือกใช้เวกเตอร์แทน SIMD ใน RISC-V

คอมพิวเตอร์แบบเวกเตอร์รวบรวมวัตถุจากหน่วยความจำหลัก และใส่ลงในเรจิสเตอร์เวกเตอร์ที่ยาวและต่อเนื่องกัน หน่วยประมวลผลแบบไปป์ไลน์ขนาน (pipelined execution units) คำนวณได้อย่างมีประสิทธิภาพสูงบนรีจิสเตอร์เวกเตอร์เหล่านี้ สถาปัตยกรรมเวกเตอร์จะกระจายผลลัพธ์จากเรจิสเตอร์เวกเตอร์กลับไปยังหน่วยความจำหลัก ขนาดของเรจิสเตอร์เวกเตอร์ถูกกำหนดโดยการออกแบบของผู้พัฒนา แทนที่จะถูกกำหนดตายตัวในรหัสดำเนินการ (opcode) เช่นเดียวกับในกรณีของ SIMD ดังที่เราจะเห็นต่อไป *การแยกความยาวเวกเตอร์และจำนวนการดำเนินการสูงสุดต่อรอบนาฬิกาออกจากการเข้ารหัสดำเนินการเป็นหัวใจของสถาปัตยกรรมเวกเตอร์:* สถาปนิกด้านไมโครเวกเตอร์ (vector microarchitect) สามารถออกแบบฮาร์ดแวร์สำหรับการประมวลผลแบบขนาน (data-parallel) ได้อย่างยืดหยุ่น โดยไม่ส่งผลกระทบต่อโปรแกรมเมอร์ และโปรแกรมเมอร์สามารถใช้ประโยชน์จากเวกเตอร์ที่มีขนาดยาวขึ้นได้โดยไม่ต้องเขียนโค้ดใหม่ ดังนั้น โค้ดไบนารีจึงสามารถใช้งานได้ข้ามการออกแบบเวกเตอร์ที่แตกต่างกัน และปรับอัตโนมัติตามความยาวเวกเตอร์ ซึ่งต่างจากชุดคำสั่ง SIMD นอกจากนี้ สถาปัตยกรรมเวกเตอร์มีคำสั่งน้อยกว่าสถาปัตยกรรมแบบ SIMD มาก และรองรับรูปแบบการคอมไพล์ที่สง่างามกว่า

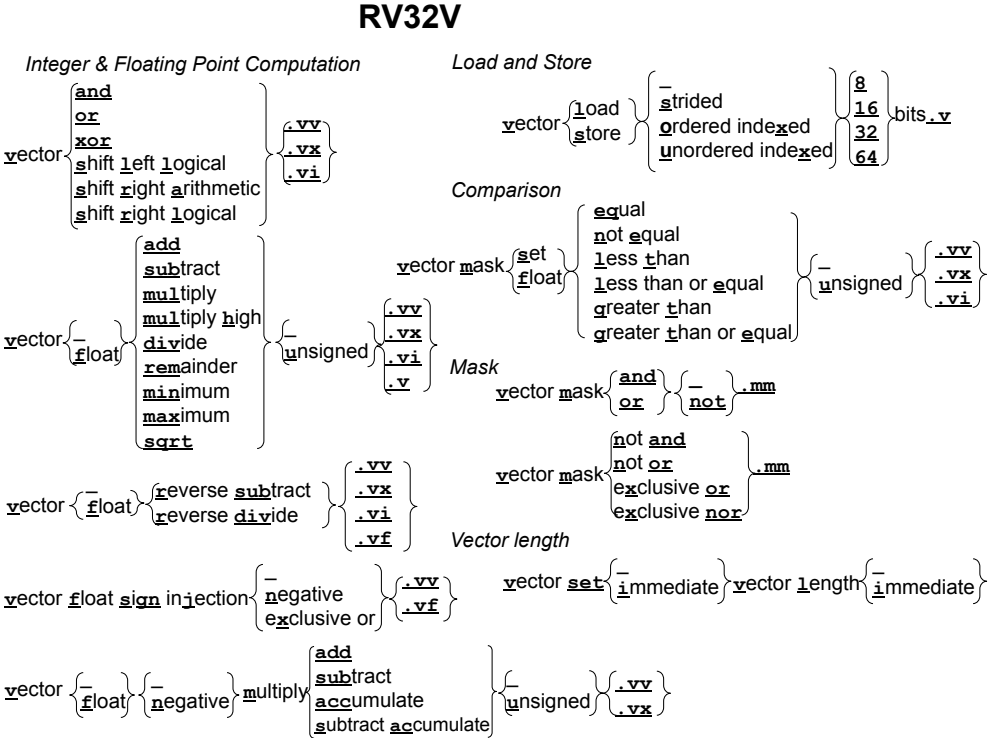
บทนี้มีการเปลี่ยนแปลงมากที่สุดในฉบับที่สองเมื่อเทียบกับบทก่อนหน้า เพราะ RV32V ยังอยู่ระหว่างการพัฒนาในปี 2017 ส่วนหนึ่งของโอกาสจากชุดคำสั่งแบบเปิดคือ สถาปนิกหลายคนสามารถมีส่วนร่วมในการออกแบบ และพวกเขานำมุมมองที่แตกต่างกันเกี่ยวกับกลยุทธ์การนำไปใช้งานมาสู่การอภิปราย มุมมองที่แตกต่างเหล่านี้นำไปสู่การเปลี่ยนแปลงที่สำคัญ ซึ่งทำให้ ISA เวกเตอร์ทั้งง่ายต่อการนำไปใช้งานและมีประสิทธิภาพมากขึ้น รูปที่ 8.1 เป็นการแสดงภาพกราฟิกของคำสั่งหลักของ RV32V

สถาปัตยกรรมเวกเตอร์ (vector architectures) นั้นพบได้น้อยกว่าสถาปัตยกรรม SIMD ดังนั้น ผู้อ่านจึงมีจำนวนน้อยที่รู้จักสถาปัตยกรรมเวกเตอร์ (หากคุณต้องการเจาะลึกเกี่ยวกับสถาปัตยกรรมเวกเตอร์ ให้อ่านบทที่ 4 และภาคผนวก G ของ [Hennessy and Patterson 2019]) บทนี้จะมีลักษณะเป็นบทแนะนำมากกว่าบทก่อนหน้า RV32V มีคุณสมบัติใหม่ที่ลดจำนวนคำสั่งใน ISA ซึ่งต้องการคำอธิบายเพิ่มเติม แม้ว่าคุณจะคุ้นเคยกับสถาปัตยกรรมเวกเตอร์อยู่แล้ว


ความเรียบง่าย

ด้วยเป้าหมายของบทนี้ บทนี้จึงเป็นบทยาวที่สุด ดังนั้นเราจะให้ภาพรวมของหัวข้อย่อยเพื่อเป็นแนวทางให้กับผู้อ่าน

- หัวข้อ 8.2 ทบทวนคำสั่งการคำนวณเวกเตอร์พื้นฐาน สำหรับข้อมูลจำนวนเต็มและจุดทศนิยม
- หัวข้อ 8.3 อธิบายถึงรีจิสเตอร์เวกเตอร์ และกำหนดประเภทแบบไดนามิก (dynamic typing) ที่แปลกใหม่ของของรีจิสเตอร์เวกเตอร์ ซึ่งความกว้างของเวกเตอร์ไม่ได้ถูกกำหนดโดย รหัสคำสั่ง (opcode) อย่างที่พบในในสถาปัตยกรรมเวกเตอร์ส่วนใหญ่ รวมถึงคุณสมบัติสำคัญสองประการของสถาปัตยกรรมเวกเตอร์ของ RISC-V: การตั้งค่าความยาวเวกเตอร์ที่จุดเริ่มต้นของแต่ละลูปโดยใช้คำสั่ง `vsetvli`—เพื่อลดค่า



รูปที่ 8.1: แผนภาพของคำสั่งเวกเตอร์ RV32V ที่ได้รับความนิยมมากที่สุด แม้ว่าโค้ดส่วนใหญ่จะใช้เพียงคำสั่งเหล่านี้ รูปที่ 8.5 ถึง รูปที่ 8.8 แสดงรูปแบบทั้งหมดของคำสั่งเหล่านี้ เช่นเดียวกับแผนภาพคำสั่งในบทก่อนหน้า รูปนี้แสดง superset ของ RV32V หมายความว่าไม่รวมทุกการผสมของคำสั่งทั้งหมด ตัวอย่างเช่น มีคำสั่ง `vfadd` และ `vaddu` แต่ไม่มีคำสั่ง `vfaddu` เพราะการคำนวณจุดทศนิยมแบบไม่มีเครื่องหมายนั้นไม่สมเหตุสมผล เนื่องจากการพิมพ์แบบไดนามิก (dynamic typing) แผนภาพคำสั่งนี้ยังสามารถใช้ได้โดยไม่ต้องเปลี่ยนแปลงสำหรับ RV64V ในบทที่ 10

ใช้จ่ายในการจัดการ—และการจัดกลุ่มรีจิสเตอร์เวกเตอร์โดยใช้ LMUL—เพื่อให้สามารถรวมกันเป็นรีจิสเตอร์ที่ยาวขึ้น เมื่อโปรแกรมเมอร์ไม่ต้องการรีจิสเตอร์เวกเตอร์ทั้ง 32 ตัว เราเห็นข้อดีของสิ่งเหล่านี้สำหรับ DAXPY ในหัวข้อ 8.8

- หัวข้อ 8.4 แนะนำคำสั่งการถ่ายโอนข้อมูลหลัก ซึ่งมีทั้งแบบก้าวหน่วย (**unit stride**), แบบก้าวข้าม (**strided**) และแบบใช้ดัชนี **indexed** เช่นเดียวกับสถาปัตยกรรมเวกเตอร์ส่วนใหญ่ นวัตกรรมหนึ่งคือ การอนุญาตให้ดำเนินการแบบใช้ดัชนีทำงานได้ทั้งแบบเรียงลำดับ (ordered) หรือไม่เรียงลำดับ (unordered)

- หัวข้อ 8.5 ชี้ให้เห็นว่า สถาปัตยกรรมเวกเตอร์ทำให้ง่ายต่อการคำนวณความกว้างของข้อมูลที่แตกต่างกัน เช่น เดียวกับในสถาปัตยกรรมคำสั่ง SIMD และขยายความยาวของเวกเตอร์ โดยไม่ต้องเปลี่ยนสถาปัตยกรรมคำสั่ง ซึ่งแตกต่างจากสถาปัตยกรรมคำสั่ง SIMD

- หัวข้อ 8.6 แสดงวิธีที่ สถาปัตยกรรมแบบเวกเตอร์จัดการคำสั่งเงื่อนไขโดยใช้รีจิสเตอร์มาสก์

- หัวข้อ 8.7 อธิบายว่า สถาปัตยกรรมคำสั่งเวกเตอร์ RISC-V ขจัดภาระการแบ่งแถบ (stripmining overhead) ที่เป็นปัญหาในสถาปัตยกรรมเวกเตอร์และ SIMD หลายตัว

- หัวข้อ 8.8 รวมส่วนต่างๆ ข้างต้นเข้าด้วยกัน เพื่อแสดงโค้ดเวกเตอร์สำหรับลูป DAXPY ซึ่งใช้คำสั่งน้อยกว่าการประมวลผลแบบ สเกลาร์มาก

- ผู้อ่านอาจต้องข้ามรายละเอียดในสิบกว่าหน้าของส่วนส่วนต่อไป และไปที่บทสรุปใน หัวข้อ 8.13 โดยตรง สำหรับผู้อ่านที่สนใจศึกษาสถาปัตยกรรมชุดคำสั่งเวกเตอร์ในเชิงลึกมากขึ้น หัวข้อ 8.9 อธิบายคำสั่งขยายและย่อขนาด (widening and narrowing instructions) สำหรับกรณีที่ตัวถูกดำเนินการปลายทาง และตัวดำเนินการต้นทางมีความกว้างไม่เท่ากัน

- หัวข้อ 8.10 อธิบายรูปแบบต่างๆ ของคำสั่งย้ายรีจิสเตอร์เวกเตอร์, การขยายจำนวนเต็ม และคำสั่งแปลงจุดทศนิยม

- หัวข้อ 8.11 สำรวจคำสั่งที่เหลือที่ใช้น้อยกว่า โดยใช้ในไลบรารีที่เขียนด้วยมือหรือเพื่อช่วยคอมไพเลอร์ปัญหาที่ซับซ้อน: การลด (reductions), การดำเนินการบนมาสก์, การจัดการองค์ประกอบภายในรีจิสเตอร์เวกเตอร์, การคำนวณแบบจุดคงตัว และคำสั่งการถ่ายโอนข้อมูลพิเศษ

- สำหรับผู้ชื่นชอบ ISA หัวข้อ 8.12 เปรียบเทียบผลลัพธ์โค้ด DAXPY สำหรับ x86 AVX2 กับ SIMD 128 บิต, x86 AVX-512 กับ SIMD 512 บิต, สถาปัตยกรรมเวกเตอร์ ARMv8 SVE และ RV64V

- หัวข้อ 8.13 สรุปด้วยการสะท้อนถึงวิวัฒนาการจาก SIMD ไปสู่สถาปัตยกรรมชุดคำสั่งเวกเตอร์

8.2 คำสั่งการคำนวณเวกเตอร์

คำสั่งเวกเตอร์ทั้งหมดขึ้นต้นด้วยตัวอักษร v การคำนวณจำนวนเต็มและทศนิยมแทบทุกคำสั่ง จากบทก่อนหน้ามีคำสั่งเวกเตอร์: รูปที่ 8.1 สืบทอดการดำเนินการจาก RV32I, RV32M, RV32H, RV32F และ RV32D คำสั่งเวกเตอร์แต่ละประเภทมีหลายรูปแบบ ขึ้นอยู่กับว่าตัวถูกดำเนินการต้นทางเป็น เรจิสเตอร์เวกเตอร์ทั้งหมด (`.vv` ต่อท้าย) รีจิสเตอร์เวกเตอร์และรีจิสเตอร์สเกลาร์จำนวนเต็ม (`.vx` ต่อท้าย) รีจิสเตอร์เวกเตอร์และค่าคงที่สเกลาร์จำนวนเต็ม (`.vi` ต่อท้าย) หรือรีจิสเตอร์เวกเตอร์และรีจิสเตอร์สเกลาร์จุดทศนิยม (`.vf` ต่อท้าย) ตัวอย่างเช่น โปรแกรม DAXPY (รูปที่ 5.5 หน้าที่ 62 บทที่ 5) คำนวณ $Y = a \times X + Y$ โดยที่ X และ Y เป็นเวกเตอร์ และ a เป็นสเกล่าร์ จึงใช้คำสั่ง `vfmacc.vf` สำหรับการดำเนินการระหว่างเวกเตอร์และรีจิสเตอร์สเกล่าร์ ฟิลด์ *rs1* ระบุรีจิสเตอร์สเกล่าร์ที่ต้องการเข้าถึง ส่วนการดำเนินการระหว่างเวกเตอร์และค่าคงที่สเกลาร์จะเข้ารหัสค่าคงที่ 5 บิต

รีจิสเตอร์ต้นทางที่หนึ่งและที่สองจะถูกสลับในรูปแบบคำสั่งสำหรับคำสั่งเวกเตอร์ส่วนใหญ่ การเก็บตำแหน่งที่อยู่ในรีจิสเตอร์สเกลาร์ที่ระบุในฟิลด์ rs1 ช่วยลดความซับซ้อนของไปป์ไลน์หน่วยความจำของฮาร์ดแวร์ การดำเนินการระหว่างเวคเตอร์และสเกลาร์ต้องการให้ตัวถูกดำเนินการที่สอง เป็นรีจิสเตอร์สเกลาร์ ดังนั้นฟิลด์ rs1 จะกลายเป็นตัวดำเนินการที่สอง และฟิลด์ rs2 จะระบุตัวถูกดำเนินการแรก

แม้ว่าค่าคงที่ (immediate) ที่มีขนาดใหญ่กว่าจะมีประโยชน์ แต่รูปแบบคำสั่งเวกเตอร์มีพื้นที่จำกัด ดังนั้นค่าคงที่จึงถูกจำกัดไว้ที่ 5 บิตของฟิลด์ระบุรีจิสเตอร์ ค่าคงที่จะถูกขยายเครื่องหมาย ยกเว้นเมื่อมันเข้ารหัสจำนวนเลื่อนบิต (shift amounts) หรือตำแหน่งองค์ประกอบ ซึ่งในกรณีนี้จะถูกขยายศูนย์ในทุกกรณี *rs2* เป็นรีจิสเตอร์เวกเตอร์ต้นทาง และ *rd* เป็นรีจิสเตอร์เวกเตอร์ปลายทาง เพื่อประหยัดพื้น opcode คำสั่งการคูณและสะสม (multiply-accumulate) ใน RVV เป็นแบบล้างค่า ดังนั้น *rd* ยังทำหน้าที่เป็นตัวถูกดำเนินการต้นทางที่สามด้วย

การดำเนินการที่ไม่สมมาตรบางอย่าง เช่น การลบและการหาร มีรูปแบบคำสั่งที่ตัวถูกดำเนินการแรกเป็นสเกลาร์และตัวที่สองเป็นเวกเตอร์ คำสั่งเหล่านี้มีคำนำหน้า `vr` หรือ `vfr` ซึ่งบ่งบอกถึงการดำเนินการแบบย้อนกลับ

ผู้อ่านอาจสังเกตว่า คำสั่งส่วนใหญ่ในรูปที่ 8.1 ไม่ระบุความกว้าง (width) ขององค์ประกอบ (elements) ที่คำสั่งนั้นดำเนินการ หัวข้อถัดไปจะอธิบายเหตุผล

■ ***รายละเอียดเพิ่มเติม: คำสั่งเวกเตอร์สามารถเปรียบเทียบเท่ากับการวนลูปในภาษา C***

ตัวอย่างเช่น โค้ดต่อไปนี้ คือสิ่งที่คำสั่ง `vadd.vv v1,v2,v3` ทำโดยพื้นฐาน

```
for (i=0;i<vl;i++)
    v1[i]=v2[i]+v3[i]
```

8.3 รีจิสเตอร์เวกเตอร์และการพิมพ์แบบไดนามิก

RV32V เพิ่มรีจิสเตอร์เวกเตอร์ 32 ตัว ซึ่งชื่อเริ่มต้นด้วย `v` จำนวนบิตในรีจิสเตอร์เวก

เตอร์ เรียกว่า *VLEN* จะแตกต่างกันไปในแต่ละเครื่อง VLEN สำหรับคอมพิวเตอร์ RISC-V ทั่วไปในปัจจุบัน VLEN จะอยู่ระหว่าง 128 ถึง 512 บิต ความกว้างของรีจิสเตอร์เวกเตอร์กำหนดขอบเขตสูงสุดของจำนวนการดำเนินการ ที่สามารถทำได้ต่อคำสั่งเวกเตอร์ ดังนั้นผู้ออกแบบโปรเซสเซอร์จะเลือก VLEN ตามพื้นที่ชิปที่สามารถเจัดสรรให้กับรีจิสเตอร์เวกเตอร์ และเป้าหมายด้านประสิทธิภาพของเครื่อง ตัวอย่างเช่น หากโปรเซสเซอร์จัดสรรพื้นที่ 1 KiB สำหรับรีจิสเตอร์เวกเตอร์ VLEN จะเท่ากับ 256 และรีจิสเตอร์เวกเตอร์ทั้ง 32 ตัว จะกว้างพอที่จะเก็บองค์ประกอบ 64 บิต ได้ 4 ตัว, องค์ประกอบ 32 บิตได้ 8 ตัว, องค์ประกอบ 16 บิตได้ 16 ตัว, หรือองค์ประกอบ 8 บิตได้ 32 ตัว

องค์ประกอบสามารถมีความกว้างได้ถึง *ELEN* บิต มาตรฐาน RVV กำหนดให้ ELEN = 64 เสมอ แต่ระบบฝังตัว (embedded systems) ใช้ ISA ย่อย (subset) ที่รองรับเพียง ELEN = 32

เพื่อรักษาความยืดหยุ่นของความกว้างของรีจิสเตอร์เวกเตอร์ใน ISA แบบเวกเตอร์ โปรเซสเซอร์เวกเตอร์จะคำนวณ *ความยาวเวกเตอร์สูงสุด* (VLMAX) ที่โปรแกรมใช้เพื่อทำงานได้อย่างถูกต้องบนโปรเซสเซอร์ ที่มีความกว้างรีจิสเตอร์เวกเตอร์ต่างกัน ความยาวนี้ถูกระบุในหน่วยจำนวนองค์ประกอบ เรจิสเตอร์ความยาวเวกเตอร์ (`vl`) จะกำหนดจำนวนองค์ประกอบในเวกเตอร์สำหรับการดำเนินการเฉพาะ ซึ่งช่วยโปรแกรมเมื่อมิติของอาร์เรย์ไม่ใช่จำนวนเต็มเท่าของ VLMAX โดย `vl` เป็นรีจิสเตอร์สถาปัตยกรรมแยกต่างหาก ที่ควบคุมคำสั่งเวกเตอร์ เราจะแสดงรายละเอียดเพิ่มเติมเกี่ยวกับ VLMAX และ `vl` ในส่วนต่อไป

เครื่อง CRAY-1 มีรีจิสเตอร์ความยาวเวกเตอร์ แต่ เป็น IBM System/370-XA ที่นำเสนอโมเดลการเขียนโปรแกรมที่ไม่ขึ้นกับ VLMAX

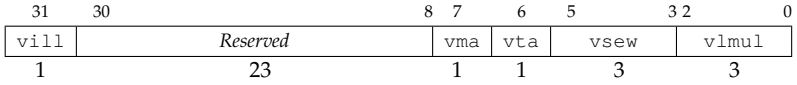


การเขียนโปรแกรม

RV32V ใช้แนวทางใหม่ โดยการเลือกความกว้างขององค์ประกอบเวกเตอร์แบบไดนามิก แทนการกำหนดไว้ตายตัวในรหัสดำเนินการก่อนการคำนวณเวกเตอร์ โปรแกรมจะตั้งค่า ความกว้างขององค์ประกอบที่เลือก *selected element width* หรือ SEW ซึ่งระบุว่ารีจิสเตอร์เวกเตอร์ถูกแบ่งเป็นองค์ประกอบอย่างไร โดยใช้ตัวอย่างข้างต้นที่ VLEN เท่ากับ 256 หาก SEW เป็น 8 รีจิสเตอร์เวกเตอร์ทั้ง 32 ตัวจะเก็บองค์ประกอบ 8 บิต ได้ตัวละ 32 องค์ประกอบ แต่ถ้า SEW เป็น 64 รีจิสเตอร์จะเก็บองค์ประกอบ 64 บิตได้ตัวละ 4 องค์ประกอบ การกำหนดประเภทรีจิสเตอร์เวกเตอร์แบบไดนามิก (dynamically typing) ช่วยลดจำนวนคำสั่งเวกเตอร์ลง ซึ่งสำคัญเพราะเครื่อง RV32V รองรับจำนวนเต็ม 3 ขนาดและจำนวนจุดทศนิยม 4 ขนาด ดังที่จะได้เห็นในหัวข้อ 8.12 เมื่อเปรียบเทียบกับคำสั่ง SIMD จำนวนมาก สถาปัตยกรรมเวกเตอร์ที่ใช้การพิมพ์แบบไดนามิก ช่วยลดภาระการรับรู้ของโปรแกรมเมอร์ภาษาแอสเซมบลี

นอกเหนือจากการอนุญาตให้โปรแกรมเมอร์เลือกความกว้างขององค์ประกอบเวกเตอร์แบบไดนามิกแล้ว RV32V ยังรองรับการจัดกลุ่มรีจิสเตอร์เวกเตอร์ (grouping vector registers) ซึ่งช่วยให้โปรแกรมเมอร์ลดจำนวนรีจิสเตอร์เวกเตอร์ที่มีผลใช้งานจริง เพื่อแลกกับการเพิ่มความยาวของมัน ตัวคูณความยาวเวกเตอร์ *vector length multiplier* หรือ LMUL ช่วยให้พื้นที่การจัดเก็บรีจิสเตอร์เวกเตอร์ที่มีจำกัดถูกใช้งานอย่างมีประสิทธิภาพมากขึ้น ในการคำนวณที่ไม่ต้องการรีจิสเตอร์เวกเตอร์จำนวนมาก เวกเตอร์ที่ยาวขึ้นสามารถปรับปรุงประสิทธิภาพได้อย่างมาก โดยเฉพาะในระบบที่มีความหน่วงของหน่วยความจำสูง LMUL สามารถตั้งค่าเป็น 1, 2, 4 หรือ 8 เมื่อ LMUL เท่ากับ 1 รีจิสเตอร์เวกเตอร์ทั้ง 32 ตัว จะพร้อมใช้งาน และแต่ละตัวมีความยาว VLEN บิต เมื่อ LMUL เท่ากับ


ประสิทธิภาพ



รูปที่ 8.2: รูปแบบรีจิสเตอร์เวกเตอร์ RV32V คือ `vtype` สามบิตทางขวาสุด เข้ารหัสตัวคูณความยาวเวกเตอร์ (vector length multiplier) หรือ LMUL เป็นเลขชี้กำลังแบบเลขฐานสองเติมเต็ม (two's complement exponent): LMUL = $2^{\texttt{vlmul}}$ สามบิตถัดมาเข้ารหัสความกว้างขององค์ประกอบที่เลือก (selected element width) หรือ SEW : SEW = $8 \times 2^{\texttt{vsew}}$ บิต สองฟิลด์ข้อมูลหนึ่งบิตถัดไประบุว่า องค์ประกอบส่วนท้าย (tail elements - `vta`) หรือองค์ประกอบที่ถูกมาสก์ (masked elements - `vma`) จะถูกเก็บรักษาไว้ (preserved, 0) หรือถูกล้าง (destroyed,1) เมื่อถูกเขียน ฟิลด์ทางซ้ายสุด คือ `vill` ระบุว่า `vtype` ถูกกำหนดเป็นค่าที่ไม่ถูกต้องหรือไม่ หาก `vill` = 1 การพยายามดำเนินการคำสั่งเวกเตอร์จะทำให้เกิดข้อยกเว้น (exception)

LMUL ยังสามารถตั้งค่าเป็นค่าเศษส่วนได้ ระหว่าง $\frac{1}{8}$ ถึง $\frac{1}{2}$ ซึ่งจะลด VLMAX ลงตามสัดส่วน แต่ไม่เพิ่มจำนวนรีจิสเตอร์เวกเตอร์ที่พร้อมใช้งาน

8 จะมีเพียงชื่อรีจิสเตอร์เวกเตอร์ทุกๆ ตัวที่แปด (`v0`, `v8`, `v16` และ `v24`) เท่านั้นที่ใพร้อมช้งาน แต่ละตัวจะมีความยาว 8×VLEN บิต โดยทั่วไป รีจิสเตอร์เวกเตอร์จะมีความยาว LMUL×VLEN บิต ดังนั้น VLMAX=LMUL×VLEN/SEW องค์ประกอบ LMUL เป็นคุณสมบัติเฉพาะของ RVV และดังที่เราจะเห็นในหัวข้อ 8.12 มันเป็นข้อได้เปรียบที่สำคัญเหนือสถาปัตยกรรมเวกเตอร์และ SIMD อื่นๆ

รีจิสเตอร์ `vtype` เดียวเข้ารหัสการตั้งค่า SEW และ LMUL สำหรับรีจิสเตอร์เวกเตอร์ทั้ง 32 ตัว รูปที่ 8.2 แสดงโครงสร้างของ `vtype` คำสั่ง `vsetvl`, `vsetvli` และ `vsetivli` ซึ่งตั้งค่ารีจิสเตอร์ความยาวเวกเตอร์ `vl` จะตั้งค่า `vtype` ด้วย รูปที่ 8.3 สรุปตัวบ่งชี้รีจิสเตอร์แบบไดนามิก ที่แนะนำในส่วนนี้

8.4 โหลดเวกเตอร์และบันทึกเวกเตอร์

กรณีที่ง่ายที่สุดสำหรับการโหลดและบันทึกเวกเตอร์แบบเวกเตอร์ คือ การจัดการกับอาร์เรย์มิติเดียวที่จัดเก็บอย่างต่อเนื่องในหน่วยความจำ คำสั่งการโหลดเวกเตอร์แบบก้าวหน่วย (unit-stride) เช่น `vle8.v`, `vle16.v`, `vle32.v` และ `vle64.v` จะเติมรีจิสเตอร์เวกเตอร์ด้วยข้อมูล จากที่อยู่ในหน่วยความจำที่ต่อเนื่องกัน โดยเริ่มจากตำแหน่งที่อยู่ในรีจิสเตอร์จำนวนเต็ม *rs1* ชื่อที่แท้จริงของ `vle8.v` คือ โหลดเวกเตอร์, องค์ประกอบ 8 บิต (**Vector Load, 8-bit Elements**); `vle16.v` คือ โหลดเวกเตอร์, องค์ประกอบ 16 บิต (**Vector Load, 16-bit Elements**) และต่อไปตามลำดับ รีจิสเตอร์ความยาวเวกเตอร์ `vl` จะกำหนดจำนวนองค์ประกอบที่จะโหลด

ตัวอย่างเช่น หาก `a0` มีค่า 1024 แล้วคำสั่ง `vle32.v v8, (a0)` จะโหลดองค์ประกอบ 32 บิตจากตำแหน่งที่อยู่ 1024, 1028, 1032, ..., 1024 + 4×(`vl`-1) ลงใน `v8` คำสั่งบันทึกเวกเตอร์แบบก้าวหน่วย (unit-stride) เช่น `vse8.v`, `vse16.v`, `vse32.v` และ `vse64.v`, จะคัดลอกรีจิสเตอร์เวกเตอร์ ไปยังตำแหน่งที่อยู่ต่อเนื่องในหน่วยความจำ

Name	Meaning	Type	Book section
`vl`	Current Vector Length in elements	Register	`8.3`
`vtype`	Vector TYPE	Register	`8.3`
`vsew`	Selected Element Width	Field of `vtype` register	`8.3`
`vlmul`	Length MULtiplier	Field of `vtype` register	`8.3`
`vta`	Tail elements preserved or not	Field of `vtype` register	`8.6`
`vma`	Mask elements preserved or not	Field of `vtype` register	`8.6`
VLEN	Size of one vector register in bits	Per-machine constant	`8.3`
SEW	Selected Element Width	Value represented by `vsew`	`8.3`
LMUL	Length MULtiplier	Value represented by `vlmul`	`8.3`
ELEN	Element LENgth: maximum vector element size	Per machine constant	`8.3`
EEW	Effective Element Width (usually EEW=SEW)	Derived value	`8.9`
EMUL	Effective register length MULtiplier	Derived value	`8.9`

รูปที่ 8.3: ชื่อและความหมายของตัวบ่งชี้ความยาวและความกว้างของเวกเตอร์ ชื่อตัวพิมพ์เล็ก (lower case names) ระบุรีจิสเตอร์หรือฟิลด์ของรีจิสเตอร์ และชื่อตัวพิมพ์ใหญ่ (upper case names) โดยทั่วไปเป็นค่าคงที่ที่คอมไพเลอร์ใช้ ค่าของตัวบ่งชี้ตัวพิมพ์ใหญ่ (upper case descriptors) แสดงถึงสิ่งที่โปรแกรมเมอร์คิด และค่าของตัวบ่งชี้ตัวพิมพ์เล็ก (lower case descriptors) เป็นการเข้ารหัสของตัวเลือกนั้นในรีจิสเตอร์ ตัวอย่างเช่น ถ้า `vsew` = 0 แล้ว `SEW` = 8 และถ้า `vlmul` = 0 ดังนั้น `LMUL` = 1 ตัวบ่งชี้บางตัวจะอธิบายเพิ่มเติม ในหัวข้อ 8.9

ที่ต่อเนื่องกันในลักษณะเดียวกัน

โปรดทราบว่า ต่างจากคำสั่งการคำนวณ คำสั่งเหล่านี้เข้ารหัสความกว้างขององค์ประกอบ (element width) ไว้ในรหัสดำเนินการ (opcode) แทนการใช้ SEW ด้วยเหตุผลที่เราจะเห็นในหัวข้อ 8.9 สำหรับตอนนี้ สมมติว่าความกว้างขององค์ประกอบที่เข้ารหัสใน opcode มีค่าเท่ากับ SEW

สำหรับอาร์เรย์หลายมิติ การเข้าถึงบางส่วนจะไม่ต่อเนื่องกัน หากจัดเก็บในลำดับแบบแถวหลัก (row major order) การเข้าถึงคอลัมน์ต่อเนื่องในอาร์เรย์สองมิติจะต้องการข้อมูลองค์ประกอบที่ถูกแยกออกจากกันตามขนาดของแถว (size of the row) สถาปัตยกรรมแบบเวกเตอร์รองรับการเข้าถึงเหล่านี้ด้วยคำสั่งโหลดแบบ (*ก้าว ข้าม*) (strided) `vlse8.v`, `vlse16.v`,`vlse32.v`, และ `vlse64.v`, และคำสั่งบันทึก เช่น (`vsse8.v`, `vsse16.v`, `vsse32.v`, และ `vsse64.v`) โดย `vlse8.v` หมายถึง *Vector Load Strided, 8-bit Elements* (โหลดเวกเตอร์แบบก้าวข้าม, องค์ประกอบ 8 บิต) คำสั่งเหล่านี้ระบุรีจิสเตอร์ต้นทางสองตัว โดย *rs1* ให้ตำแหน่งที่อยู่เริ่มต้น และ *rs2* ระบุระยะก้าว (stride) ในหน่วยไบต์

ตัวอย่างเช่น สมมติว่า `a0` มีตำแหน่งที่อยู่เริ่มต้นใน 1024 และ `a1` มีขนาดแถว 64 ไบต์ คำสั่ง `vlse32.v v8, (a0), a1` จะส่งลำดับตำแหน่งที่อยู่ต่อไปนี้ไปยังหน่วยความจำ: 1024, 1088, 1152, ..., 1024 + 64×(`vl`-1) ข้อมูลที่ส่งกลับจะถูกเขียนลงในองค์ประกอบ 32 บิต ที่ต่อเนื่องกันของรีจิสเตอร์เวกเตอร์ปลายทาง

การเข้าถึงหน่วยความจำแบบมีระยะห่าง (strided accesses) เป็นการขยายจากการ



เข้าถึงแบบระยะห่างหน่วย **unit-stride**: สามารถสร้างผลลัพธ์ของ `vle32.v` ได้โดยใช้ `vlse32.v` ด้วยระยะห่าง 4 ไบต์ แต่การเลือกคำสั่งแรก จะรับประกันต่อโปรเซสเซอร์ทราบว่า การเข้าถึงเป็นแบบต่อเนื่อง ทำให้โปรเซสเซอร์สามารถให้แบนด์วิดท์หน่วยความจำสูงได้ง่ายขึ้น การทำเช่นนี้ยังลดขนาดโค้ดสำหรับกรณีการเข้าถึงระยะห่างหน่วย (unit-stride) ที่พบบ่อย เพราะไม่ต้องโหลดระยะห่างลงใน *rs2*

การเขียนโปรแกรม

จนถึงตอนนี้ เราได้สมมติว่าโปรแกรมกำลังทำงานกับอาร์เรย์แบบหนาแน่น (dense arrays) เพื่อรองรับอาร์เรย์บาง (sparse arrays) สถาปัตยกรรมเวกเตอร์มีคำสั่งโหลดแบบ*ดัชนี* (indexed loads) เช่น (`vloxei8.v`, `vloxei16.v`, `vloxei32.v` และ `vloxei64.v`) และคำสั่งบันทึก (`vsoxei8.v`, `vsoxei16.v`, `vsoxei32.v` และ `vsoxei64.v`) รีจิสเตอร์ต้นทางหนึ่งตัวสำหรับคำสั่งเหล่านี้จะอ้างถึงรีจิสเตอร์เวกเตอร์ และรีจิสเตอร์อีกหนึ่งตัวอ้างถึงรีจิสเตอร์สเกลาร์ รีจิสเตอร์สเกลาร์ *rs1* มีตำแหน่งที่อยู่เริ่มต้นของอาร์เรย์บาง และองค์ประกอบแต่ละตัวของรีจิสเตอร์เวกเตอร์ *rs2* มีดัชนี (index) ในหน่วยไบต์ขององค์ประกอบที่ไม่เป็นศูนย์ของอาร์เรย์บาง สำหรับการโหลดและบันทึกแบบดัชนี รหัสดำเนินการจะ เข้ารหัสความกว้างขององค์ประกอบดัชนี ในขณะที่ SEW จะระบุความกว้างขององค์ประกอบข้อมูลที่ถูกโหลดหรือบันทึก ด้วยวิธีนี้ เวกเตอร์ดัชนีและเวกเตอร์ข้อมูลสามารถใช้ขนาดองค์ประกอบที่แตกต่างกันได้ ซึ่งมักเป็นกรณีนี้

สมมติว่า `a0` มีตำแหน่งที่อยู่เริ่มต้นที่ 1024 และรีจิสเตอร์เวกเตอร์ `v4` มีดัชนีในหน่วยไบต์ในองค์ประกอบ 4 ตัวแรกดังนี้: 16, 48, 80, 160 คำสั่ง `vloxei32.v v0,a0,v4` จะส่งลำดับตำแหน่งที่อยู่นี้ไปยังหน่วยความจำ: 1040 (1024 + 16), 1072 (1024 + 48), 1104 (1024 + 80), 1184 (1024 + 160) คำสั่งนี้จะวางข้อมูลที่โหลดมาไว้ในองค์ประกอบที่ต่อเนื่องกัน โดยแต่ละองค์ประกอบมีความกว้าง SEW บิต ในรีจิสเตอร์เวกเตอร์ปลายทาง

การโหลดแบบดัชนี (indexed load) ยังเรียกว่า *รวบรวม* (gather) และการบันทึกแบบใช้ดัชนีมักเรียกว่า *กระจาย* (scatter)

ประสิทธิภาพ

RISC-V มีคำสั่งถ่ายโอนข้อมูลเวกเตอร์แบบดัชนีสองแบบ: *แบบเรียงลำดับ* และ *แบบไม่เรียงลำดับ* คำสั่งแบบไม่เรียงลำดับบอกโปรเซสเซอร์ว่าการโหลดและการบันทึกองค์ประกอบแต่ละตัวสามารถเกิดขึ้นในลำดับใดก็ได้และยังคงให้ผลลัพธ์ที่ต้องการ ซึ่งอาจนำไปสู่แบนด์วิดท์หน่วยความจำที่สูงขึ้นในบางการนำไปใช้งาน ตัวอย่างเช่น การพลาดแคช (cache misses) อาจส่งผลต่อเวลาที่การเข้าถึงหน่วยความจำบางส่วนเกิดขึ้น เนื่องจากเป็นไปได้ที่ดัชนีเดียวกันอาจปรากฏหลายครั้ง คำสั่งแบบเรียงลำดับ (ordered) จะบอกโปรเซสเซอร์ว่าเพื่อให้ได้ผลลัพธ์ที่ถูกต้อง การเข้าถึงต้องเกิดขึ้นตามลำดับขององค์ประกอบ ตัวอย่างเช่น ดัชนีที่ซ้ำกันตัวสุดท้ายในเรจิสเตอร์เวกเตอร์ของคำสั่งบันทึกแบบใช้ดัชนีที่เรียงลำดับ ควรเป็นตัวกำหนดค่าในหน่วยความจำ โดยทั่วไป คอมไพเลอร์จะสร้างคำสั่งการถ่ายโอนข้อมูลแบบใช้ดัชนีที่เรียงลำดับ เว้นแต่มันจะสามารถพิสูจน์ได้ว่าดัชนีทั้งหมดไม่ซ้ำกัน คำสั่งภาษาแอสเซมบลีใช้ตัวอักษร *o* สำหรับ ordered แบบเรียงลำดับ—เช่น `vloxei32.v` หมายถึง Vector Load Ordered indeXed, 32-bit Elements *(การโหลดเวกเตอร์แบบเรียงลำดับด้วยดัชนี องค์ประกอบ 32 บิต)*—และใช้ตัวอักษร *u* สำหรับ unordered แบบที่ไม่เรียงลำดับ—เช่น `vluxei32.v` หมายถึง Vector Load Unordered indeXed, 32-bit Elements *(การโหลดเวกเตอร์แบบไม่เรียงลำดับด้วยดัชนี*

องค์ประกอบ 32 บิต)

อาเรย์บาง (Sparse arrays) เป็นแรงจูงใจของเราในการโหลดและบันทึกแบบใช้ดัชนี แต่มีอัลกอริธึมอื่นๆ อีกมากมาย ที่เข้าถึงข้อมูลทางอ้อมผ่านตารางดัชนี

8.5 การทำงานแบบขนานในระหว่างการดำเนินการเวกเตอร์

ในขณะที่โปรเซสเซอร์เวกเตอร์แบบง่ายอาจประมวลผลองค์ประกอบเวกเตอร์ทีละตัว การดำเนินการขององค์ประกอบแต่ละตัวโดยนิยามแล้วเป็นอิสระต่อกัน ดังนั้น ในทางทฤษฎี โปรเซสเซอร์สามารถคำนวณองค์ประกอบทั้งหมดพร้อมกันได้ ข้อมูลที่มีความกว้างสูงสุดใน RV32G คือ 64 บิต และโปรเซสเซอร์เวกเตอร์ในปัจจุบันมักประมวลผลองค์ประกอบ 64 บิตได้ 2, 4, หรือ 8 ตัวต่อรอบสัญญาณนาฬิกา ฮาร์ดแวร์จัดการกรณีขอบ (fringe cases) เมื่อความยาวเวกเตอร์ไม่เป็นทวีคูณของจำนวนองค์ประกอบ ที่ประมวลผลต่อรอบสัญญาณนาฬิกา


ประสิทธิภาพ

เช่นเดียวกับ SIMD จำนวนการดำเนินการข้อมูลขนาดเล็กจะเพิ่มขึ้น เมื่อความกว้างขององค์ประกอบลดลง ดังนั้น โปรเซสเซอร์เวกเตอร์ที่คำนวณการดำเนินการ 64 บิต 4 ตัวต่อรอบนาฬิกาจะสามารถเริ่มการดำเนินการ 32 บิตได้ 8 ตัว, 16 บิตได้ 16 ตัว, และ 8 บิตได้ 32 ตัวต่อรอบนาฬิกา

ใน SIMD สถาปนิกของ ISA เป็นผู้กำหนดจำนวนสูงสุดของการดำเนินการแบบข้อมูลคู่ขนานต่อคำสั่ง *และ* จำนวนองค์ประกอบต่อรีจิสเตอร์ ในทางตรงกันข้าม นักออกแบบโปรเซสเซอร์ RV32V สามารถเลือกทั้งสองอย่างนี้โดยไม่ต้องเปลี่ยนแปลง ISA คอมไพเลอร์ หรือแม้แต่โค้ดไบนารี ในขณะที่การเพิ่มความกว้างของเรจิสเตอร์ SIMD เป็นสองเท่าจะทำให้จำนวนคำสั่ง SIMD เพิ่มขึ้นมากกว่าสองเท่า และต้องมีการการเปลี่ยนแปลงในคอมไพเลอร์ SIMD รวมถึงต้องคอมไพล์โปรแกรมใหม่ ความยืดหยุ่นที่ซ่อนอยู่นี้หมายความว่าโปรแกรม RV32V เดียวกัน สามารถทำงานได้อย่างมีประสิทธิภาพโดยไม่ต้องแก้ไขใดๆ บนโปรเซสเซอร์เวกเตอร์ทั้งแบบที่ง่ายที่สุดและแบบที่ล้ำสมัยที่สุด


การเขียนโปรแกรม


ประสิทธิภาพ

8.6 การดำเนินการแบบมีเงื่อนไขของการคำนวณเวกเตอร์

โค้ดเวกเตอร์บางส่วนมีการใช้คำสั่ง `if` โปรแกรมเมอร์สามารถใช้การกระโดดแบบมีเงื่อนไขเพื่อกำหนดค่าให้องค์ประกอบเวกเตอร์แต่ละตัว แต่จะไม่มีประสิทธิภาพ นับตั้งแต่เครื่อง Cray-1 สถาปัตยกรรมแบบเวกเตอร์ได้รวมมาสก์ (mask) ที่ระงับ (suppresses) การดำเนินการบนองค์ประกอบบางตัวของการดำเนินการเวกเตอร์ คำสั่งเปรียบเทียบ ในรูปที่ 8.1 จะทำการทดสอบเงื่อนไขระหว่างเวกเตอร์สองตัว หรือระหว่างเวกเตอร์กับสเกลาร์ และเขียนผลลัพธ์ลงใน แต่ละบิตของรีจิสเตอร์ปลายทางเป็น 1 หากเงื่อนไขเป็นจริง หรือ 0 หากไม่เป็นจริง คำสั่งเวกเตอร์ต่อมาสามารถใช้มาสก์นั้นได้ โดยบิต 1 ในตำแหน่งที่ *i* หมายถึงองค์ประกอบที่ *i* จะถูกเปลี่ยนแปลงโดยการดำเนินการเวกเตอร์ และไม่เปลี่ยนแปลงหากบิตเป็น 0 เพื่อระบุว่าการดำเนินการใช้มาสก์ โปรแกรมเมอร์ภาษาแอสเซมบลีจะเพิ่ม

`.t` ต่อท้ายคำสั่ง

ใน RV32V คำสั่งสามารถเป็นแบบไม่ใช้มาสก์ (ซึ่งหมายถึง ดำเนินการแบบไม่มีเงื่อนไข) หรือใช้มาสก์โดยรีจิสเตอร์เวกเตอร์ตัวแรก (`v0`) เมื่อคำสั่งใช้มาสก์ บิตที่มีนัยสำคัญน้อยที่สุดของแต่ละองค์ประกอบใน `v0` จะระบุว่าการดำเนินการขององค์ประกอบนั้นใช้งานอยู่ (active) หรือไม่ คำสั่ง `vmand`, `vmnand`, `vmandn`, `vmxor`, `vmor`, `vmnor`, `vmorn` และ `vmxnor` จะดำเนินการทางตรรกะบนมาสก์ที่เก็บในรีจิสเตอร์เวกเตอร์ เพื่อให้สามารถประมวลผลคำสั่งเงื่อนไขที่ซ้อนกันได้อย่างมีประสิทธิภาพ

การมีความยืดหยุ่นมากขึ้น ในการเลือกรีจิสเตอร์สำหรับมาสก์จะทำให้การเขียนโปรแกรมง่ายขึ้น และลดความกดดันในการใช้รีจิสเตอร์มาสก์ แต่การใช้เพียงรีจิสเตอร์เดียว (`v0`) จะช่วยให้การเข้ารหัสคำสั่งมีขนาดกะทัดรัด ทำให้คำสั่งเวกเตอร์สามารถเข้ารหัสได้ใน 32 บิต

ตัวอย่าง เช่น สมมติว่าองค์ประกอบที่มีเลขลำดับคู่ในรีจิสเตอร์เวกเตอร์ `v3` เป็นจำนวนเต็มลบ และองค์ประกอบที่มีเลขลำดับคี่เป็นจำนวนเต็มบวก ผลลัพธ์ของโค้ดนี้

```
vslt.vx  v0, v3, x0        # ตั้งค่าบิตใน v0 ถึง 1 เมื่อสมาชิกใน v3 < 0
vadd.vv  v4, v1, v2, v0.t  # เปลี่ยนสมาชิกใน v4 ถึง v1+v2 v0 ตามมาส์ก
```

จะตั้งค่าบิตที่มีเลขลำดับคู่ใน `v0` เป็น 1 และบิตที่มีเลขลำดับคี่เป็น 0 และจะแทนที่องค์ประกอบที่มีเลขลำดับคู่ใน `v4` ด้วยผลรวมขององค์ประกอบที่สอดคล้องกันใน `v1` และ `v2`

โปรแกรม จะ เรียก ว่า *สามารถทำให้เป็นเวกเตอร์ได้* หากการดำเนินการส่วนใหญ่ใช้คำสั่งเวกเตอร์ คำสั่ง gather, scatter และคำสั่งที่ใช้มาสก์ จะเพิ่มจำนวนโปรแกรมที่สามารถทำให้เป็นเวกเตอร์ได้

แล้วองค์ประกอบที่มีเลขลำดับคี่ใน `v4` จะเป็นอย่างไร? โปรแกรมเมอร์สามารถระบุได้ว่าองค์ประกอบที่มีเลขลำดับคี่ใน `v4` ควรคงค่าเดิมไว้หรือเนื้อหานั้นไม่สำคัญ และโปรเซสเซอร์สามารถทิ้งได้ (discard) โดยตั้งค่าฟิลด์ `vma` ในรีจิสเตอร์ `vtype` เป็น 0 หรือ 1 ตามลำดับ



ประสิทธิภาพ



การเขียนโปรแกรม

รีจิสเตอร์ `vtype` มีการควบคุมการเปลี่ยนแปลงที่คล้ายกัน สำหรับเนื้อหาขององค์ประกอบส่วนท้าย (tail elements) ของเวกเตอร์เมื่อ `vl` < VLMAX การตั้งค่าฟิลด์ `vta` เป็น 0 ระบุว่าองค์ประกอบส่วนท้ายควรคงค่าเดิมไว้ และการตั้งค่าเป็น 1 ระบุว่าโปรแกรมเมอร์ไม่สนใจเนื้อหาของมัน และดังนั้นโปรเซสเซอร์สามารถทิ้งได้ตามดุลยพินิจ RVV มีตัวเลือกทั้งแบบ “ไม่สนใจเนื้อหา” (agnostic) และ “คงค่าเดิม” (undisturbed) สำหรับองค์ประกอบที่ใช้มาสก์และองค์ประกอบส่วนท้าย เพราะตัวเลือกแรกบางครั้งช่วยปรับปรุงประสิทธิภาพของโปรเซสเซอร์ ในขณะที่ตัวเลือกหลังช่วยในการทำให้เป็นเวกเตอร์ (vectorization)

8.7 การควบคุมความยาวเวกเตอร์

ดังที่ได้กล่าวไว้ก่อนหน้านี้ คำสั่ง `vsetvl`, `vsetvli` และ `vsetivli` จะตั้งค่าทั้งความยาวเวกเตอร์ (`vl`) และ `vtype` ซึ่งรวมถึงความกว้างขององค์ประกอบที่เลือก หรือ SEW, ตัวคูณความยาวเวกเตอร์สูงสุด หรือ LMUL และการตั้งค่าแบบไม่สนใจส่วนท้าย (tail-agnostic) และไม่สนใจมาสก์ (mask-agnostic) (`vta` และ `vma`) คำอธิบายในโค้ดด้านล่างแสดงวิธีการทำงานของคำสั่ง `vset` แต่ละตัว

```
vsetvli  rd, rs1, vtypei  # rd = vl ใหม่, rs1  = AVL, vtypei = ค่า vtype ใหม่
vsetivli rd, uimm, vtypei # rd = vl ใหม่, uimm = AVL, vtypei = ค่า vtype ใหม่
vsetvl   rd, rs1, rs2     # rd = vl ใหม่, rs1  = AVL, rs2    = ค่า vtype ใหม่
```

ทั้ง `vsetvl` และ `vsetvli` จะรับความยาวเวกเตอร์ที่แอปพลิเคชันร้องขอ (ความยาวเวกเตอร์ที่ใช้งานได้) (**application vector length** หรือ AVL) จากรีจิสเตอร์จำนวนเต็ม *rs1* ในขณะที่ `vsetivli` รับจากค่าคงที่ AVL (จาก *rs1* หรือค่าคงที่) และ VLMAX จะกำหนดความยาวเวกเตอร์ใหม่ `vl` เครื่องมีอิสระบางประการในการเลือก `vl` ใหม่ แต่จะเลือกค่าไม่เกิน AVL หรือ VLMAX (การนำไปใช้งานส่วนใหญ่ตั้ง `vl` เป็นค่าที่น้อยกว่าระหว่าง AVL และ VLMAX นั่นคือ `vl` = min(AVL,VLMAX)) ความยาวเวกเตอร์ `vl` ใหม่จะถูกเขียนลงในรีจิสเตอร์จำนวนเต็ม *rd* ด้วย การตั้งค่า `vtype` ใหม่จะกำหนด VLMAX ใหม่

ระบบ `vsetvl` ช่วยให้ซอฟต์แวร์ประมวลผลอาร์เรย์เป็นหน่วยขนาด VLMAX โดยไม่ต้องรู้ค่า VLMAX ในเวลาคอมไพล์ และไม่ขึ้นกับขนาดของอาร์เรย์จะหารด้วย VLMAX ลงตัวหรือไม่ ในทุกการวนรอบ หากเหลือองค์ประกอบ `n` ตัวให้ประมวลผล ซอฟต์แวร์จะเรียกคำสั่ง `vsetvl` เพื่อร้องขอเวกเตอร์ขนาด `n` องค์ประกอบ เครื่องจะให้เวกเตอร์ขนาด `n` องค์ประกอบ หรือขนาด `vl` ที่น้อยกว่า `n` จากนั้น ซอฟต์แวร์จะดำเนินการคำสั่งเวกเตอร์ที่ประกอบเป็นเนื้อหาของรอบสุดท้าย ซอฟต์แวร์จะลบ `vl` ออกจาก `n` และทำซ้ำรอบจนกว่า `n` จะถึงศูนย์ ณ จุดนี้ อาร์เรย์ทั้งหมดจะถูกประมวลผลเรียบร้อย การแบ่งอาร์เรย์เป็นส่วนๆ ขนาด ที่เครื่องจัดการได้และวนรอบในลักษณะนี้เรียกว่า การแบ่งส่วนข้อมูล *(strip-mining)* เราจะเห็นตัวอย่างของเทคนิคสำคัญนี้ ในหัวข้อต่อไป

■ ***รายละเอียดเพิ่มเติม:*** `vsetvl`, `vsetvli` ***มีรูปแบบพิเศษ เมื่อ*** **rs1** ***เป็น*** `x0`

คำอธิบายข้างต้นใช้สำหรับกรณีทั่วไปที่ *rs1* ไม่ใช่ `x0` มีรูปแบบพิเศษสองรูปแบบที่ช่วยเร่งกรณีการใช้งานที่พบน้อยลงดังนี้:

- หาก *rs1* เป็น `x0` แต่ *rd* ไม่ใช่ `x0` คำสั่งจะเปลี่ยน `vtype` ตามปกติ จากนั้นตั้ง `vl` และ *rd* เป็น VLMAX
- หาก *rs1* และ *rd* เป็น `x0` ทั้งคู่ คำสั่งจะเปลี่ยน `vtype` ตามปกติ แต่คง `vl` เดิมไว้ รูปแบบนี้ถูกต้องเฉพาะเมื่อ `vl` เดิมไม่เกิน VLMAX ใหม่ เพื่อรับประกันคุณสมบัตินี้ การรักษาอัตราส่วนของ SEW ต่อ LMUL ให้คงที่ก็เพียงพอ เพราะจะทำให้ VLMAX คงที่

8.8 ตัวอย่างเวกเตอร์ DAXPY ใน RV32V

แม้ว่า RVV จะมีฟีเจอร์เพิ่มเติมอีกมาก แต่ตอนนี้เราได้เห็นเพียงพอที่จะอธิบายตัวอย่างครบถ้วนแล้ว รูปที่ 8.4 แสดงภาษาแอสเซมบลี RV32V สำหรับ DAXPY (รูปที่ 5.5 ในหน้าที่ 62 บทที่ 5) ซึ่งเราจะอธิบายทีละขั้นตอน

ในโค้ด DAXPY ของ RV32V ไม่มี prologue เราจะเข้าสู่ลูปหลักทันที ขั้นตอนแรกคือการใช้คำสั่ง `vsetvli` เพื่อตั้งค่า `vl` และ `vtype` โดยเราจะเน้นที่ `vtype` ก่อน `vtype` ประกอบด้วยความกว้างขององค์ประกอบที่เลือก (SEW), ตัวคูณความยาวเวกเตอร์สูงสุด (LMUL) และการตั้งค่าแบบไม่สนใจส่วนท้าย (tail-agnostic) และไม่สนใจมาสก์ (mask-agnostic) เราตั้งค่า SEW เป็น 64 บิต เพราะ DAXPY ทำงานกับจำนวนจุดทศนิยมความแม่นยำสองเท่า เนื่องจาก DAXPY ต้องการรีจิสเตอร์เวกเตอร์เพียงไม่กี่ตัว—แค่สองตัว เพื่อ

```
# a0 คือ n, a1 ตัวชี้ไปที่ x[0], a2 ตัวชี้ไปที่ y[0], fa0 คือ a
loop:
  0: 0db576d7 vsetvli a3, a0, e64,m8,ta,ma   # avl=n; SEW=64; LMUL=8
  4: 0205f007 vle64.v v0, (a1)               # load vector x
  8: 02067407 vle64.v v8, (a2)               # load vector y
  c: b2055457 vfmacc.vf v8, fa0, v0          # v8 += fa0 * v0
 10: 02067427 vse64.v v8, (a2)               # store vector y
 14: 8d15     sub     a0, a0, a3            # n -= vl
 16: 068e     slli    a3, a3, 3             # vl * 8 (bytes)
 18: 95b6     add     a1, a1, a3            # เพิ่ม x pointer
 1a: 9636     add     a2, a2, a3            # เพิ่ม y pointer
 1c: f175     bnez    a0, loop              # repeat if n != 0
 1e: 8082     ret                           # return
```

รูปที่ 8.4: โค้ด RV32V สำหรับ DAXPY ในรูปที่ 5.5 คำสั่งสเกลาร์ ในรูปที่ 5.7 หน้าที่ 65 มีความคล้ายคลึงกันมาก โดยมีจำนวนและประเภทของคำสั่งเท่ากัน ยกเว้น `vsetvli` ที่มาแทนที่การกระโดดแบบมีเงื่อนไขแรก

เก็บส่วนของอาร์เรย์ `x` และ `y`—เราสามารถตั้งค่า LMUL เป็นค่าสูงสุดที่ 8 ซึ่งช่วยให้ใช้เวกเตอร์ที่ยาวขึ้น แต่จำกัดเราให้ใช้กลุ่มละ 8 รีจิสเตอร์เวกเตอร์ เราจะใช้ `v0` เพื่อเก็บ `x` และ `v8` เพื่อเก็บ `y` สุดท้าย เราระบุว่าเราไม่สนใจค่าในองค์ประกอบที่มีมาสก์และองค์ประกอบส่วนท้ายในโปรแกรมนี้ เมื่อรวมทุกอย่างเข้าด้วยกัน เราใช้ไวยากรณ์ `e64,m8,ta,ma` เพื่อระบุ SEW=64, LMUL=8, การไม่สนใจส่วนท้าย (tail-agnosticism) และการไม่สนใจมาสก์ (mask-agnosticism) ตามลำดับ

สมมติว่าโปรเซสเซอร์ RV32V ของเรามีสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์ ขนาด 2048 ไบต์ (เราเลือกขนาดนั้นเพื่อให้ตรงกับ x86 SIMD ISA ที่เปรียบเทียบใน หัวข้อ 8.12) เนื่องจากมีรีจิสเตอร์เวกเตอร์ 32 ตัว แต่ละตัวมีขนาด 64 ไบต์ แต่เมื่อเราได้ตั้งค่า LMUL เป็น 8 รีจิสเตอร์เวกเตอร์ที่เหลือทั้งสี่ตัวจะกลายเป็น 512 ไบต์ หรือ 64 องค์ประกอบความแม่นยำสองเท่า ดังนั้น VLMAX สำหรับสำหรับลูปนี้คือ 64

นอกเหนือจากการตั้งค่า `vtype` คำสั่ง `vsetvli` ที่ส่วนบนของลูป ยังตั้งค่าความยาวเวกเตอร์สำหรับคำสั่งเวกเตอร์ที่ตามมา แต่ละการวนร้องขอความยาวเวกเตอร์เท่ากับจำนวนองค์ประกอบที่เหลืออยู่ในอาร์เรย์ คำสั่ง `vsetvli` กำหนดจำนวนองค์ประกอบที่จะถูกประมวลผลจริง ในการวนนี้—บางจำนวนระหว่าง `n` และ VLMAX รวมทั้งสองค่า ถ้า `n` มีค่ามาก ในรอบการวนส่วนใหญ่ `vl` จะเท่ากับ VLMAX เมื่อลูปใกล้เสร็จสิ้น หาก `n` ไม่หารด้วย VLMAX ลงตัว `vl` จะน้อยกว่า VLMAX ซึ่งป้องกันไม่ให้เครื่องอ่านหรือเขียนเกินขอบเขตของ `x` และ `y`

สถาปัตยกรรมเวกเตอร์ที่ไม่มี `vl` ต้องการโค้ดพิเศษเพื่อจัดการกับองค์ประกอบสุดท้ายของลูป และเพื่อตรวจสอบว่า `n` เป็นศูนย์ในตอนแรกหรือไม่

คำสั่ง `vle64.v` ที่ตำแหน่งที่อยู่ `4` เป็นโหลดเวกเตอร์จากตำแหน่งที่อยู่ของ `x` ในรีจิสเตอร์สเกลาร์ `a1` โดยถ่ายโอน `vl` องค์ประกอของ `x` จากหน่วยความจำไปยัง `v0`

คำสั่ง `vle64.v` ที่ตำแหน่งที่อยู่ `8` โหลด `vl` องค์ประกอบของ `y` จากหน่วยความจำไปยัง `v8` คำสั่งถัดไป (`add`) เพิ่มตัวชี้ไปยัง `x`

คำสั่งตำแหน่งที่อยู่ `c` คือจุดเด่น `vfmacc.vf` คูณ `vl` องค์ประกอบของ `x` (`v0`) ด้วยสเกลาร์ `a` (`fa0`) และบวกผลคูณแต่ละตัวเข้ากับ `vl` องค์ประกอบของ `y` (`v8`) และเก็บผล

โปรเซสเซอร์ เวก เตอร์ CRAY X1 จริงๆ แล้วปรับแต่งการวน*สองครั้งสุดท้าย*ให้เหมาะสม หากเหลือ 17 องค์ประกอบ และเครื่องเวกเตอร์ดำเนินการได้ 16 ครั้งต่อการวน มันจะแบ่งการวนสองครั้งสุดท้ายเป็น 9 และ 8 องค์ประกอบ เพราะจะทำงานได้มีประสิทธิภาพมากกว่า 16 และ 1 RISC-V ยังอนุญาต (แต่ไม่บังคับ) การปรับแต่งเช่นนี้ในคำจำกัดความของ `vsetvli`

รวม `vl` ตัว นั้นกลับไปยัง `y` (`v8`)

สิ่งที่เหลือคือการจัดเก็บผลลัพธ์ลงในหน่วยความจำและงานบางส่วนของลูป คำสั่งจัดเก็บที่ที่อยู่ 10 (`vse64.v`) จัดเก็บผลลัพธ์ `vl` ตัวลงใน `y` ในหน่วยความจำ คำสั่งตำแหน่งที่อยู่ `14` (`sub`) ลดค่า `n` (`a0`) ลงด้วย `vl` เพื่อบันทึกจำนวนการดำเนินการที่เสร็จสิ้นในการวนลูปนี้ คำสั่งเลื่อน (shift) (`slli`) ถัดไปคูณความยาวเวกเตอร์ด้วยความกว้างของข้อมูลในหน่วยไบต์ (8) เพื่อใช้ในการเพิ่มตัวชี้ไปยัง `x` และ `y` ในภายหลัง คำสั่ง `add` ที่ตำแหน่งที่อยู่ `16` และ `18` เพิ่มตัวชี้ไปยัง `x` และ `y` ตามลำดับ คำสั่งกระโดดที่ตำแหน่งที่อยู่ `1a` ทำการวนลูปซ้ำถ้า `n` (`a0`) ไม่เป็นศูนย์ ถ้า `n` เป็นศูนย์ คำสั่งสุดท้าย `ret` จะคืนการควบคุมไปยังจุดที่เรียกใช้งาน

พลังของสถาปัตยกรรมเวกเตอร์ คือ แต่ละรอบของลูป 10 คำสั่งนี้ เริ่มต้นการเข้าถึงหน่วยความจำ 3 × 64 = 192 ครั้ง และการดำเนินการจุดทศนิยม 2 × 64 = 128 ครั้ง (สมมติว่า `n` มีค่าอย่างน้อยเท่ากับ VLMAX) โดยเฉลี่ยประมาณ 19 การเข้าถึงหน่วยความจำและ 13 การดำเนินการจุดทศนิยมต่อคำสั่ง ดังที่เราจะเห็นในหัวข้อ 8.12 อัตราส่วนเหล่านี้สำหรับสถาปัตยกรรม SIMD ของ x86 ที่มีขนาดรีจิสเตอร์สถาปัตยกรรมเวกเตอร์เท่ากันนั้นด้อยกว่าถึงห้าเท่า


ประสิทธิภาพ

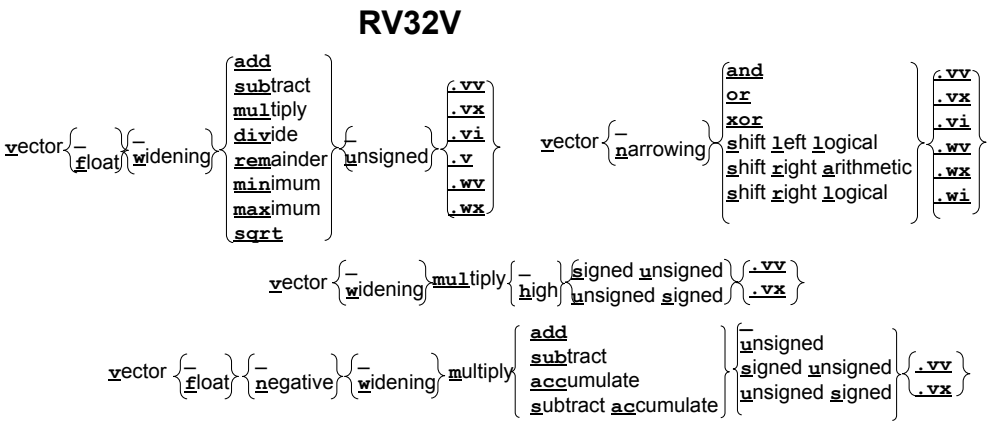
ตามที่กล่าวไว้ในบทนำ ผู้อ่านที่สนใจในคำอธิบายของคุณลักษณะหลักของ RV32V สามารถข้ามเนื้อหาหลายสิบหน้าถัดไปเพื่อไปยังหัวข้อบทสรุปในส่วน 8.13 ส่วนผู้อ่านที่สนใจภาพรวมควรดำเนินการต่อไป

8.9 คำสั่งขยายและคำสั่งแคบลง

สมมติฐานจนถึงตอนนี้ คือกรณีทั่วไปที่ตัวถูกดำเนินการต้นทางและปลายทางทั้งหมดมีความกว้างเท่ากัน ด้วยการพึ่งพาการพิมพ์แบบไดนามิก (dynamic typing) เพื่อลดจำนวนคำสั่ง RVV จัดการการดำเนินการเลขคณิตที่ตัวถูกดำเนินการต้นทางและปลายทางมีความกว้างต่างกันอย่างไร? คำตอบคือการเสนอการดำเนินการสามประเภทเพิ่มเติม:

1. การดำเนินการเลขคณิตแบบ *ขยาย* (Widening) ที่ความกว้างขององค์ประกอบที่มีผล (Effective Element Width - EEW) ของปลายทางเป็น 2×SEW บิต และ EEW ของตัวถูกดำเนินการต้นทางทั้งสองเป็น SEW bits
2. การดำเนินการทางเลขคณิตแบบขยายที่ EEW ของปลายทางและตัวถูกดำเนินการต้นทางแรก (`vs2`) เป็น 2×SEW bits และEEW ของตัวถูกดำเนินการต้นทางที่สอง (`vs1`) เป็น SEW bits
3. การดำเนินการเลขคณิตแบบ *แคลลง (Narrowing)* ที่ EEW ของปลายทางและตัวถูกดำเนินการต้นทางที่สอง (`vs1`) เป็น SEW bits และ EEW ของตัวถูกดำเนินการต้นทางแรก (`vs2`) เป็น 2×SEW bits.

สถาปนิกพบว่ามีความจำเป็นน้อยมากสำหรับการดำเนินการแบบแคบลง ที่ตัวถูกดำเนินการทั้งสองมีความกว้าง 2×SEW bits รูปที่ 8.5 แสดงการดำเนินการทั้งหมดที่มีรุ่น

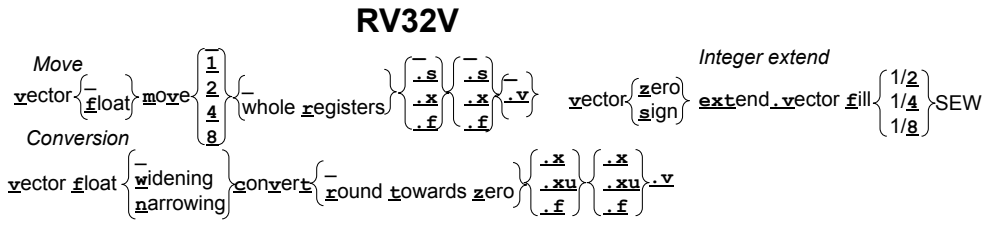


รูปที่ 8.5: แผนภาพของคำสั่งเลขคณิตและตรรกะแบบขยายและย่อขนาด ซึ่งไม่แสดงใน รูปที่ 8.1 เนื่องจากการพิมพ์แบบไดนามิก แผนภาพคำสั่งนี้ยังใช้งานได้โดยไม่ต้องเปลี่ยนแปลงสำหรับ RV64V ในบทที่ 10

ขยาย (widening) และแคบลง (narrowing) ที่ไม่แสดงใน รูปที่ 8.1 เพื่อไม่ให้ผู้อ่านรู้สึกหนักเกินไปนหัวข้อบทนำของ RV32V

หากตัวดำเนินการบางตัวมีองค์ประกอบที่กว้างกว่าตัวอื่น ๆ องค์ประกอบที่กว้างกว่านั้นจะพอดีกับรีจิสเตอร์เวกเตอร์ได้อย่างไร? คำตอบคือ ตัวดำเนินการเหล่านี้จะครอบครองรีจิสเตอร์เวกเตอร์หลายตัวราวกับว่า LMUL ถูกเพิ่มขึ้น กล่าวให้แม่นยำยิ่งขึ้น *ตัวคูณความยาวที่มีผล (Effective LMUL - EMUL)* ของแต่ละตัวถูกดำเนินการจะเท่ากับ LMUL×EEW/SEW ดังนั้น หาก LMUL ปัจจุบันเป็น 1 ปลายทางของการดำเนินการขยายจะใช้รีจิสเตอร์เวกเตอร์สองตัว เพราะ EEW ของมันเป็นสองเท่าของ SEW

การเขียนสัญลักษณ์ภาษาแอสเซมบลีสำหรับการดำเนินการที่มีความกว้างผสมใช้คำนำหน้า `vw` สำหรับคำสั่งขยายจำนวนเต็ม และ `vfw` สำหรับคำสั่งขยายจุดทศนิยม การดำเนินการแบบย่อขนาด ใช้คำนำหน้า `vn` และ `vfn` ตามลำดับ ส่วนต่อท้ายมาตรฐาน `.vv` และ `.vx` ถูกตีความว่า หมายถึงตัวถูกดำเนินการแรกของการดำเนินการแบบขยายเป็นแบบแคบ โดยใช้ `.wv` และ `.wx` เพื่อระบุว่าตัวถูกดำเนินการตัวแรกนั้นกว้างด้วย (จึงใช้ `w` แทน `v` ในส่วนต่อท้าย)



รูปที่ 8.6: แผนภาพของคำสั่งย้าย การขยายจำนวนเต็ม และการแปลงของ RV32V เนื่องจากการพิมพ์แบบไดนามิก แผนภาพคำสั่งนี้จึงยังใช้งานได้โดยไม่ต้องเปลี่ยนแปลงสำหรับ RV64V ในบทที่ 10

ตัวอย่างเช่น หาก SEW ถูกตั้งค่าสำหรับตัวถูกดำเนินการ 32 บิต คำสั่งต่อไปนี้จะสร้างผลคูณ 64 บิตจากตัวถูกดำเนินการ 32 บิตสองตัว และผลคูณ 64 บิตจากตัวถูกดำเนินการ 64 บิตและตัวถูกดำเนินการ 32 บิต:

รักษา SEW/LMUL ให้คงที่ (และทำให้ VLMAX คงที่)เมื่อดำเนินการกับเวกเตอร์ที่มีค่าความแม่นยำต่างกันโดยการปรับเปลี่ยน `vtype` แบบไดนามิก

```
vwmul.vv  v0, v2, v1   # ความกว้างของ v0 เป็น 2*SEW v2 เป็น SEW และ v1 เป็น SEW
vwmul.wv  v0, v2, v1   # ความกว้างของ v0 เป็น 2*SEW v2 เป็น 2*SEW และ v1 เป็น SEW
```

คำสั่งต่อไปนี้เลื่อนตัวถูกดำเนินการ 64 บิต ไปทางขวา และใส่ผลลัพธ์ลงในรีจิสเตอร์ปลายทาง 32 บิต โดยจำนวนการเลื่อนบิตเป็นตัวถูกดำเนินการ 32 บิต

```
vnsrl.vv  v0, v2, v1   # ความกว้างของ v0 เป็น SEW, v2 เป็น SEW และ v1 เป็น 2*SEW
```

โหลดและจัดเก็บยังยึดตามแนวคิดของ EEW และ EMUL ในหัวข้อ 8.4 เราได้กล่าวไว้ว่า การโหลดและบันทึกเข้ารหัสความกว้างขององค์ประกอบในรหัสดำเนินการ ตัวอย่างเช่น `vse16.v` เข้ารหัสว่าข้อมูลที่จัดเก็บมี EEW เป็น 16 บิต เช่นเดียวกับการดำเนินการทางเลขคณิต นี่ยังหมายความว่า EMUL ของข้อมูลที่บันทึก คือ LMUL×EEW/SEW

ต่อไปนี้คือตัวอย่างสั้นๆ ที่โหลดเวกเตอร์สองตัวของจำนวนเต็ม 8 บิต ใช้การคูณแบบขยายเพื่อสร้างเวกเตอร์จำนวนเต็ม 16 บิต จากนั้นเก็บผลลัพธ์ไว้ในหน่วยความจำ

```
vsetivli  x0, 4, e8, m1, ta, ma  # เวกเตอร์ของสิ่งค์ประกอบ 8 บิต
vle8.v    v0, (a0)               # โหลดสิ่งค์ประกอบ 8 บิต
vle8.v    v1, (a1)               # โหลดสิ่งค์ประกอบ 8 บิต
vwmul.vv  v2, v1, v0             # สร้างสิ่งค์ประกอบ 16 บิต
vse16.v   v2, (a2)               # จัดเก็บสิ่งค์ประกอบ 16 บิต
```

โปรดทราบว่า เราไม่จำเป็นต้องเปลี่ยน `vtype` หรือ `vl` ก่อนบันทึกลงในหน่วยความจำ แม้ว่าคำสั่งบันทึก จะเข้าถึงบิตในหน่วยความจำมากกว่าคำสั่งโหลดทั้งสองถึงสองเท่า

8.10 การย้ายรีจิสเตอร์เวกเตอร์, การขยายจำนวนเต็มและคำสั่งแปลง

คำสั่งประเภทนี้รวมถึงรุ่นต่างๆ ของการย้ายรีจิสเตอร์ การขยายจำนวนเต็ม และการแปลง รูปที่ 8.6 แสดงการนำเสนอภาพกราฟิกของคำสั่งเหล่านี้

คำสั่งอ่าน/เขียนสเกลาร์จำนวนเต็ม ถ่ายโอนค่าเดียวระหว่างรีจิสเตอร์ x แบบสเกลาร์และองค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ โดยขยายเครื่องหมาย หากรีจิสเตอร์ปลายทางกว้างกว่ารีจิสเตอร์ต้นทาง

```
vmv.x.s rd, vs2  # x[rd] = vs2[0] (vs1=0)
vmv.s.x vd, rs1  # vd[0] = x[rs1] (vs2=0)
```

มีการถ่ายโอนที่เทียบเท่าสำหรับรีจิสเตอร์จุดทศนิยม: `vfmv.fs` และ `vfmv.sf`


ประสิทธิภาพ

คำสั่งย้ายทั้งรีจิสเตอร์สามารถคัดลอกลุ่มของรีจิสเตอร์เวกเตอร์ โดยไม่ต้องกังวลเกี่ยวกับ `vl` หรือ `vtype`, ซึ่งสามารถเพิ่มประสิทธิภาพได้ คำสั่ง `vmv<nr>r.v` เหล่านี้สามารถคัดลอกกลุ่มของรีจิสเตอร์เวกเตอร์ 1, 2, 4, หรือ 8 ตัวอย่างเช่น:

```
vmv1r.v v1,  v2   # คัดลอก v1=v2
vmv2r.v v10, v12  # คัดลอก v10=v12; v11=v13
vmv4r.v v4,  v8   # คัดลอก v4=v8; v5=v9; v6=v10; v7=v11
vmv8r.v v0,  v8   # คัดลอก v0=v8; v1=v9; …; v7=v15
```

คำสั่งขยายจำนวนเต็มจะทำการขยายศูนย์หรือขยายเครื่องหมายจากหนึ่งในสอง, หนึ่งในสี่ หรือหนึ่งในแปดของรีจิสเตอร์ให้เป็นความกว้างเต็ม SEW EEW และ EMUL ของรีจิสเตอร์ต้นทางจะถูกปรับลดลงให้สอดคล้องกัน ตัวอย่างเช่น รีจิสเตอร์ต้นทางของ `vsext.vf4` มี EEW=SEW/4 และ EMUL=LMUL/4

```
vzext.vf2 vd,vs2,vm  # ขยายศูนย์จาก SEW/2 ถึง SEW
vsext.vf2 vd,vs2,vm  # ขยายเครื่องหมายจาก SEW/2 ถึง SEW
vzext.vf4 vd,vs2,vm  # ขยายศูนย์จาก SEW/4 ถึง SEW
vsext.vf4 vd,vs2,vm  # ขยายเครื่องหมายจาก SEW/4 ถึง SEW
vzext.vf8 vd,vs2,vm  # ขยายศูนย์จาก SEW/8 ถึง SEW
vsext.vf8 vd,vs2,vm  # ขยายเครื่องหมายจาก SEW/8 ถึง SEW
```

ตามที่คาดไว้ RVV ยังมีคำสั่งการแปลงมากมายที่แปลงระหว่างจุดทศนิยมที่มีความกว้างต่างๆ และจำนวนเต็มแบบมีเครื่องหมายหรือไม่มีเครื่องหมาย การแปลงเหล่านี้มีจำนวนมากแต่ไม่น่าแปลกใจ (ดูรายละเอียดเพิ่มเติม เกี่ยวกับการแปลงตัวเลขในหัวข้อ 5.4) ด้านล่างนี้คือตัวอย่างบางส่วน ซึ่งสามารถเข้าใจได้ด้วยตนเองจากคอมเมนต์ โปรดสังเกตการใช้ `w` และ `n` สำหรับการแปลงแบบขยายหรือย่อขนาด

```
vfcvt.xu.f.v       vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มไม่มีเครื่องหมาย
vfcvt.x.f.v        vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มมีเครื่องหมาย
vfcvt.rtz.xu.f.v   vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มไม่มีเครื่องหมาย, ปัดทิ้ง
vfcvt.rtz.x.f.v    vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มมีเครื่องหมาย, ปัดทิ้ง
vfcvt.f.xu.v       vd,vs2,vm  # จำนวนเต็มไม่มีเครื่องหมายเป็นจุดทศนิยม
vfcvt.f.x.v        vd,vs2,vm  # จำนวนเต็มมีเครื่องหมายเป็นจุดทศนิยม

vfwcvt.xu.f.v      vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มไม่มีเครื่องหมาย 2 เท่า
vfwcvt.x.f.v       vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มมีเครื่องหมาย 2 เท่า
vfwcvt.rtz.xu.f.v vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มไม่มีเครื่องหมาย 2 เท่า, ปัดทิ้ง
vfwcvt.rtz.x.f.v   vd,vs2,vm  # จุดทศนิยมเป็นจำนวนเต็มมีเครื่องหมาย 2 เท่า, ปัดทิ้ง
vfwcvt.f.xu.v      vd,vs2,vm  # จำนวนเต็มไม่มีเครื่องหมายเป็นจุดทศนิยม 2 เท่า
vfwcvt.f.x.v       vd,vs2,vm  # จำนวนเต็มมีเครื่องหมายเป็นจุดทศนิยม 2 เท่า
vfwcvt.f.f.v       vd,vs2,vm  # จุดทศนิยมเป็นจุดทศนิยม 2 เท่า

vfncvt.xu.f.w      vd,vs2,vm  # จุดทศนิยม 2 เท่าเป็นจำนวนเต็มไม่มีเครื่องหมาย
vfncvt.x.f.w       vd,vs2,vm  # จุดทศนิยม 2 เท่าเป็นจำนวนเต็มมีเครื่องหมาย
vfncvt.rtz.xu.f.w vd,vs2,vm  # จุดทศนิยม 2 เท่าเป็นจำนวนเต็มไม่มีเครื่องหมาย, ปัดทิ้ง
vfncvt.rtz.x.f.w   vd,vs2,vm  # จุดทศนิยม 2 เท่าเป็นจำนวนเต็มมีเครื่องหมาย, ปัดทิ้ง
vfncvt.f.xu.w      vd,vs2,vm  # จำนวนเต็มไม่มีเครื่องหมาย 2 เท่าเป็นจุดทศนิยม
vfncvt.f.x.w       vd,vs2,vm  # จำนวนเต็มมีเครื่องหมาย 2 เท่าเป็นจุดทศนิยม
vfncvt.f.f.w       vd,vs2,vm  # จุดทศนิยม 2 เท่าเป็นจุดทศนิยม
vfncvt.rod.f.f.w   vd,vs2,vm  # ดทศนิยม 2 เท่าเป็นจุดทศนิยม, ปัดไปยังเลขคี่
```

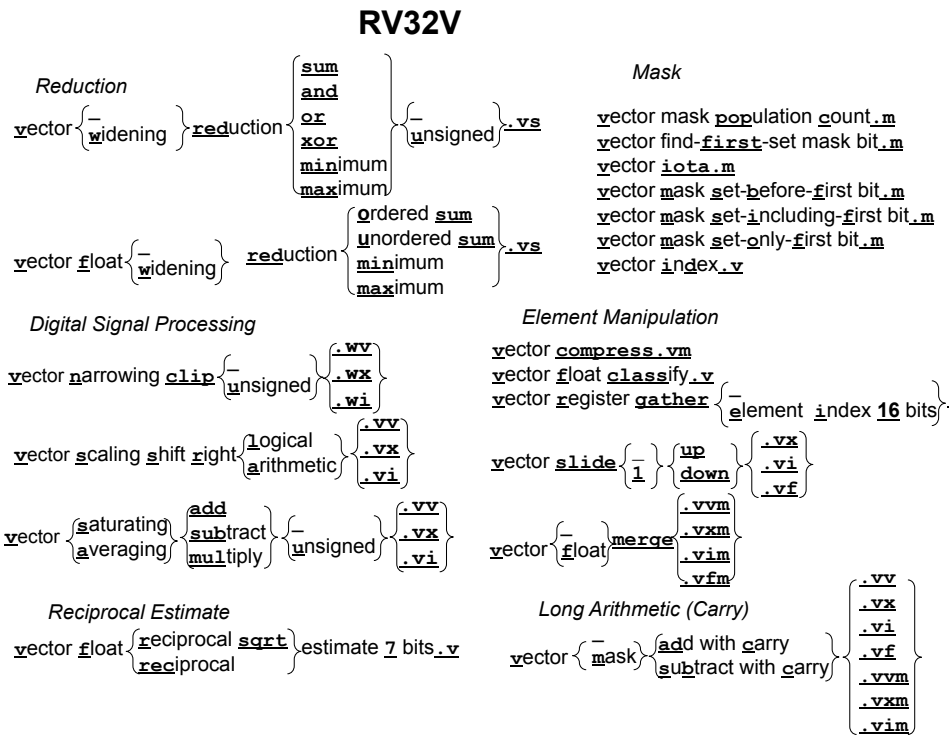
การแปลงแบบย่อขนาด มี EEW และ EMUL ของ*รีจิสเตอร์ต้นทาง*เป็นสองเท่าของ SEW และ LMUL การแปลงแบบขยาย มี EEW และ EMUL ของ*รีจิสเตอร์ปลายทาง* เป็นสองเท่า SEW และ LMUL

8.11 คำสั่งเวกเตอร์ที่เร่งความเร็วให้กับไลบรารี

ส่วนก่อนหน้านี้ อธิบายคำสั่งเวกเตอร์ที่คอมไพเลอร์สามารถสร้างจากโค้ดต้นฉบับภาษาระดับสูง (HLL) ในส่วนนี้จะแนะนำคำสั่งเวกเตอร์ ที่ออกแบบมาเพื่อใช้ในไลบรารีที่ถูกสร้างอย่างประณีตเป็นหลัก—ไม่ว่าจะสร้างขึ้นผ่านคำสั่งกำกับของคอมไพเลอร์เท่านั้น หรือเขียนด้วยภาษาแอสเซมบลีจริงๆ—แม้ว่าบางคำสั่งจะรวบรวมไว้เพื่อให้คอมไพเลอร์สร้างในสถานการณ์พิเศษ คำสั่งเหล่านี้อาจไม่ใช้ในสถานการณ์ทั่วไป แต่สามารถช่วยได้มากในสถานการณ์ที่เหมาะสม การดำเนินการเช่นนี้ยังเป็นเหตุผลของคำสั่ง SIMD นับร้อยในสถาปัตยกรรม x86 จุดเด่นที่ช่วย RVV คือความกว้างของเวกเตอร์และความกว้างของประเภทข้อมูลไม่ได้อยู่ใน opcode ดังนั้นจึงไม่มากเท่า

เราได้จัดกลุ่มคำสั่งเหล่านี้เป็นแปดหมวดหมู่

- การลดข้ามองค์ประกอบทั้งหมดของรีจิสเตอร์เวกเตอร์ตัวเดียว
- การดำเนินการระดับบิตของรีจิสเตอร์มาสก์
- การจัดการองค์ประกอบภายในรีจิสเตอร์เวกเตอร์ตัวเดียว
- การคำนวณจุดคงตัวตัวสำหรับการประมวลผลสัญญาณดิจิตอล



รูปที่ 8.7: แผนภาพของคำสั่ง RV32V สำหรับการลด, มาสก์, การประมวลผลสัญญาณดิจิตอล, การจัดการองค์ประกอบ และการคำนวณเกินความกว้างปกติที่มีบิตส่วนเกิน (carry) เนื่องจากการพิมพ์แบบไดนามิก แผนภาพคำสั่งนี้ยังทำงานโดยไม่ต้องเปลี่ยนแปลงสำหรับ RV64V ในบทที่ 10

- การประมาณส่วนกลับ
- การเรียงสับเปลี่ยน
- การคำนวณมัลติเวิร์ด
- คำสั่งการถ่ายโอนข้อมูลที่ปรับปรุงสิทธิภาพให้เหมาะสม หรือหลีกเลี่ยงข้อยกเว้นที่ไม่ตั้งใจ

รูปที่ 8.7 แสดงคำสั่งสำหรับห้าหมวดหมู่แรก


ประสิทธิภาพ

การลด การดำเนินการเวกเตอร์ที่มีประโยชน์อย่างหนึ่ง เรียกว่า *การลด* ซึ่งดำเนินการข้ามองค์ประกอบทั้งหมดของรีจิสเตอร์เวกเตอร์ RVV มีการลดที่รวมถึงผลรวม (sum), ค่าต่ำสุด (min), และค่าสูงสุด (max) สำหรับจำนวนเต็มมีเครื่องหมาย จำนวนเต็มไม่มีเครื่องหมาย และจำนวนจุดทศนิยม RVV ยังมีคำสั่งขยายการลดผลรวม สำหรับจำนวนเต็ม RVV มีการลดสำหรับการดำเนินการเชิงตรรกะ AND, OR และ XOR เนื่องจากการบวกจุดทศนิยมไม่เป็นคุณสมบัติเชิงสัมพันธ์ RVV จึงมีทั้งผลรวมจุดทศนิยมแบบเรียงลำดับและไม่เรียงลำดับ เพื่อให้ได้ประสิทธิภาพสูงขึ้น เมื่อโปรแกรมไม่สนใจลำดับ และความสามารถในการทำซ้ำได้เมื่อโปรแกรมต้องการ อินพุตสำหรับการดำเนินการลด (reduction

operations) คือองค์ประกอบทั้งหมดของรีจิสเตอร์เวกเตอร์ต้นทางแรกและองค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ต้นทางที่สอง โดยผลลัพธ์จะถูกส่งไปยังองค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ปลายทาง องค์ประกอบที่เหลือในรีจิสเตอร์เวกเตอร์ปลายทางถือเป็นส่วนท้าย (tail) และจะถูกรักษาไว้หรือไม่ขึ้นอยู่กับฟิลด์ `vta` (ดูรูปที่ 8.3) ต่อไปนี้คือตัวอย่างบางส่วน:

```
# การลดจำนวนเต็มอย่างง่าย โดยที่ [*] หมายถึงองค์ประกอบที่ใช้งานทั้งหมด
vredsum.vs  vd, vs2, vs1, vm   # vd[0] =   sum (vs1[0], vs2[*])
vredmaxu.vs vd, vs2, vs1, vm   # vd[0] =  maxu (vs1[0], vs2[*])
vredmax.vs  vd, vs2, vs1, vm   # vd[0] =   max (vs1[0], vs2[*])
vredminu.vs vd, vs2, vs1, vm   # vd[0] =  minu (vs1[0], vs2[*])
vredmin.vs  vd, vs2, vs1, vm   # vd[0] =   min (vs1[0], vs2[*])

# การลดผลรวมจำนวนเต็มแบบขยาย
vwredsumu.vs vd, vs2, vs1, vm  # vd[0] = ผลรวม (vs1[0], ขยายศูนย์ (vs2[*]))
vwredsum.vs  vd, vs2, vs1, vm  # vd[0] = ผลรวม (vs1[0], ขยายศูนย์ (vs2[*]))

# การลดทศนิยมอย่างง่าย
vfredosum.vs vd, vs2, vs1, vm  # ผลรวมแบบเรียงลำดับ
vfredusum.vs vd, vs2, vs1, vm  # ผลรวมแบบไม่เรียงลำดับ

# การลดทศนิยมแบบขยาย
vfwredosum.vs vd, vs2, vs1, vm # ผลรวมแบบเรียงลำดับ
vfwredusum.vs vd, vs2, vs1, vm # ผลรวมแบบไม่เรียงลำดับ
```

การดำเนินการมาสก์ จุดประสงค์หลักของรีจิสเตอร์มาสก์ คือเพิ่มส่วนของโค้ดที่สามารถทำให้เป็นเวกเตอร์ได้โดยจัดการการดำเนินการแบบมีเงื่อนไข เราได้กล่าวถึงคำสั่งมาสก์ที่สนับสนุนคำสั่งเงื่อนไขแบบซ้อนกันไปก่อนหน้านี้: `vmand`, `vmnand`, `vmandn`, `vmxor`, `vmor`, `vmnor`, `vmorn` และ `vmxnor`

เมื่อโปรแกรมเมอร์ใช้รีจิสเตอร์เวกเตอร์เป็นมาสก์ EEW ที่ได้สำหรับแต่ละองค์ประกอบจะเป็นเพียง 1 บิต

คำสั่งมาสก์เวกเตอร์ที่เหลือมีประโยชน์มากสำหรับอัลกอริธึมเฉพาะ ตัวอย่างแรกๆ บางตัวมีมาตั้งแต่รุ่นแรกของสถาปัตยกรรมเวกเตอร์:

- `vcpop.m` *การนับจำนวนประชากรมาสก์เวกเตอร์* นับจำนวนองค์ประกอบมาสก์ของรีจิสเตอร์มาสก์เวกเตอร์ต้นทาง ที่มีค่า 1 และเขียนผลลัพธ์ลงในรีจิสเตอร์ x แบบสเกลาร์
- `vfirst.m` *ค้นหาบิตมาสก์ที่ตั้งค่าแรก* ค้นหาองค์ประกอบที่มีหมายเลขต่ำสุดของมาสก์ต้นทางที่มีค่า 1 และเขียนดัชนีขององค์ประกอบนั้นลงในรีจิสเตอร์ x แบบสเกลาร์ หากไม่มีองค์ประกอบใดมีค่า 1 จะเขียน -1
- `viota.m` เขียนผลรวมคำนำขนานของค่ามาสก์ลงในรีจิสเตอร์เวกเตอร์ปลายทาง โดยอ่านรีจิสเตอร์มาสก์เวกเตอร์ต้นทาง และเขียนลงในแต่ละองค์ประกอบของเรจิสเตอร์เวกเตอร์ปลายทางเป็นผลรวมของบิตทั้งหมดขององค์ประกอบในเรจิสเตอร์มาสก์ที่มีดัชนีน้อยกว่าองค์ประกอบนั้น

คำสั่งมาสก์ต่อไปนี้สร้างรูปแบบบิตที่โดยทั่วไปมี 1 จำนวนมากตามด้วย 0 จำนวนมาก—ซึ่งโดยทั่วไปเรียกว่ามาสก์—โดยอิงจากตำแหน่งของบิตมาสก์แรกที่เป็น 1

- `vmsbf.m` *ตั้งค่าก่อนบิตมาสก์แรก* อ่านและเขียนรีจิสเตอร์มาสก์ แล้วเขียน 1 ลงในองค์ประกอบมาสก์ทั้งหมด *ก่อน* องค์ประกอบต้นทางแรกที่เป็น 1 (นับจากขวาไปซ้าย) จากนั้นเขียน 0 ลงในองค์ประกอบนั้น และองค์ประกอบที่ตามมาทั้งหมดที่ตามมา หากไม่มีบิตใดในเวกเตอร์ต้นทางเป็น 1 บิตมาสก์ปลายทางทั้งหมดจะได้ค่า 1
- `vmsif.m` *ตั้งค่ารวมถึงบิตหน้ากากแรก* คล้ายกับ `vmsbf.m` โดยเปลี่ยนแปลงเพียงอย่างเดียวคือมันเขียนค่า 1 ลงในตำแหน่งที่ตรงกันด้วย รวมถึงบิตมาสก์ก่อนหน้านั้น
- `vmsof.m` *ตั้งค่าเฉพาะบิตมาสก์แรก* คล้ายกับ `vmsbf.m` แต่ตั้งค่าเฉพาะองค์ประกอบแรกของปลายทางเป็น 1 ส่วนองค์ประกอบที่เหลือทั้งหมดถูกตั้งเป็น 0

สุดท้าย `vid.v`, *ดัชนีองค์ประกอบเวกเตอร์* เขียนดัชนีของแต่ละองค์ประกอบลงในรีจิสเตอร์เวกเตอร์ปลายทาง ตั้งแต่ 0 ถึง `vl`-1 ตัวอย่างการใช้ `vid.v` หนึ่ง คือการประเมินฟังก์ชันทางคณิตศาสตร์บนเวกเตอร์ของอาร์กิวเมนต์ที่เว้นระยะห่างเชิงเส้น: ในกรณีนี้อาร์กิวเมนต์เป็นฟังก์ชันเชิงเส้นสัมพันธ์ (affine function) ของ ID องค์ประกอบ ตัวอย่างอีกหนึ่งคือการสนับสนุนโมเดลการเขียนโปรแกรมแบบ CUDA-like ซึ่งโปรแกรมเมอร์ระบุการคำนวณอย่างชัดเจนในแง่ของ ID องค์ประกอบ

การคำนวณจุดคงตัว การดำเนินการเหล่านี้ พบได้ทั่วไปในการประมวลผลสัญญาณดิจิทัล คำสั่ง `vnclip` และคำสั่ง `vnclipu` บีบอัดค่าจุดคงตัวแบบมีเครื่องหมาย หรือไม่มีเครื่องหมาย ลงในปลายทางที่ย่อขนาดโดยใช้การปัดเศษ การปรับสเกล และการอิ่มตัวขณะที่ลดขนาดข้อมูล คำสั่งการปรับสเกล `vssra` และคำสั่ง `vssrl` เลื่อนค่าอินพุตไปทางขวา และปัดเศษบิตที่เลื่อนออกตามรีจิสเตอร์สถานะ `vxrm` คำสั่งบวกและลบแบบเฉลี่ย `vaadd`, `vaaddu`, `vasub`, และ `vasubu` เลื่อนผลลัพธ์ไปทางขวาหนึ่งบิตและปัดเศษผลลัพธ์ตาม `vxrm` คำสั่งบวกและลบแบบอิ่มตัว `vsadd`, `vsaddu`, `vssub` และ `vssubu` ไม่วนกลับเมื่อเกิดโอเวอร์โฟลว์ แต่จะสร้างค่าต่ำสุดหรือสูงสุดสำหรับประเภทข้อมูลนั้นแทน คำสั่งคูณแบบอิ่มตัว `vsmul` คล้ายกัน แต่ยังเลื่อนผลคูณที่มีขนาด 2×SEW บิตไปทางขวา SEW-1 บิต และปัดเศษตาม `vxrm` ก่อนที่จะอิ่มตัวลงในผลคูณขนาด SEW บิต

การประมาณค่าส่วนกลับ RV32V มีคำสั่งสองคำสั่งที่ประมาณค่า $1/x$ หรือ $1/\sqrt{x}$ ด้วยความแม่นยำ 7 บิต: `vfrec7.v` และ `vfrsqrt7.v` คำสั่งเหล่านี้มักทำงานเร็วกว่าคำสั่งหารเวกเตอร์ `vfdiv.vv` หรือรากที่สองเวกเตอร์ `vfsqrt.v` บางครั้งโปรแกรมเมอร์ใช้คำสั่งเหล่านี้ตามด้วยการทำซ้ำวิธี Newton-Raphson เพื่อเพิ่มความแม่นยำเป็นสองเท่า


ประสิทธิภาพ

การจัดเรียง RV32V มีคำสั่งหลายคำสั่งที่จัดเรียงองค์ประกอบภายในเรจิสเตอร์เวกเตอร์ *การรวบรวมรีจิสเตอร์เวกเตอร์* (`vrgather.vv`) สร้างเวกเตอร์ผลลัพธ์ใหม่ โดยการรวบรวมองค์ประกอบจากเวกเตอร์ข้อมูลต้นทางหนึ่งตาม ตำแหน่งองค์ประกอบที่ระบุโดยเวกเตอร์ดัชนีต้นทางที่สอง

```
# อาร์กิวเมนต์สุดท้ายเก็บค่าจาก 0..VLMAX-1
# ซึ่งเลือกองค์ประกอบจาก vs2
vrgather.vv     vd, vs2, vs1, vm  # เวกเตอร์ของดัชนีใน vs1
vrgather.vx     vd, vs2, rs1, vm  # ดัชนีเดี่ยวใน x[rs1]
vrgather.vi     vd, vs2, rs1, vm  # ดัชนีค่าคงที่เดี่ยว
vrgatherei16.vv vd, vs2, vs1, vm  # เวกเตอร์ของดัชนี EEW=16 ใน vs1
```

ดังนั้น ถ้าองค์ประกอบสี่ตัวแรกของ `v2` มีค่า 8, 0, 4, 2 แล้ว `vrgather.vv v0,v1,v2` จะแทนที่องค์ประกอบที่ศูนย์ของ `v0` ด้วยองค์ประกอบที่แปดของ `v1` องค์ประกอบที่หนึ่งของ `v0` ด้วยองค์ประกอบที่ศูนย์ของ `v1` องค์ประกอบที่สองของ `v0` ด้วยองค์ประกอบที่สี่ของ `v1`, และองค์ประกอบที่สามของ `v0` ด้วยองค์ประกอบที่สองของ `v1` หากดัชนีเกิน VLMAX-1 องค์ประกอบปลายทางที่สอดคล้องกันจะถูกตั้งเป็นศูนย์

`vrgather` ยังมีรูปแบบ `.vx` และ `.vi` ซึ่งดึงองค์ประกอบหนึ่งตัวจากเวกเตอร์ต้นทาง และทำซ้ำองค์ประกอบนั้นไปทั่วทั้งเวกเตอร์ปลายทาง รูปแบบเหล่านี้ทำงานเร็วกว่า `vrgather.vv` อย่างมากในหลายการออกแบบ

`vrgatherei16.vv` คล้ายกับ `vrgather.vv` แต่เวกเตอร์ดัชนีมี EEW=16 ทำให้ข้อมูลและดัชนีมีความกว้างต่างกันได้

การผสานเวกเตอร์ (`vmerge.vvm`) ผสานรีจิสเตอร์เวกเตอร์สองตัวเข้าด้วยกัน มันสร้างเวกเตอร์ผลลัพธ์ใหม่โดยเลือกแต่ละองค์ประกอบจากเรจิสเตอร์ต้นทางสองตัว ขึ้นอยู่กับเรจิสเตอร์มาสก์ องค์ประกอบใหม่จะมาจาก vs2 หากบิตมาสก์เป็น 0 หรือจาก vs1 หากบิตมาสก์เป็น 1:

```
# องค์ประกอบ i ของ v0 กำหนดว่าองค์ประกอบ i ใหม่สำหรับ vd มาจาก
# vs2[i] ถ้าบิตมาสก์  i ของ v0 เป็น 0 หรือมาจาก
vmerge.vvm vd, vs2, vs1, v0  # vs1[i] ถ้าบิตมาสก์ i ของ v0 เป็น 1
vmerge.vxm vd, vs2, rs1, v0  # x[rs1] ถ้าบิตมาสก์ i ของ v0 เป็น 1
vmerge.vim vd, vs2, imm, v0  # imm    ถ้าบิตมาสก์ i ของ v0 เป็น 1
```

ดังนั้น หากองค์ประกอบสี่ตัวแรกของ `v0` มีค่า 1, 0, 0, 1, องค์ประกอบสี่ตัวแรกของ `v2` มีค่า 1, 2, 3, 4 และองค์ประกอบสี่ตัวแรกของ `v1` มีค่า 10, 20, 30, 40 แล้วคำสั่ง `vmerge.vvm v3,v2,v1,v0` จะทำให้องค์ประกอบสี่ตัวแรกของ `v3` มีค่า 10, 2, 3, 40

คำสั่ง `vmerge` ยังมีรูปแบบ `.vxm` และ `.vim` ซึ่งคัดลอกจากรีจิสเตอร์จำนวนเต็มหรือค่าคงที่ที่ขยายเครื่องหมาย เมื่อบิตมาสก์เป็น 1

คำสั่ง *การเลื่อนลงของเวกเตอร์* `vslidedown` นำองค์ประกอบที่เริ่มจากตำแหน่งกลางของเวกเตอร์หนึ่ง และวางไว้ที่จุดเริ่มต้นของรีจิสเตอร์เวกเตอร์ที่สอง

```
vslidedown.vx vd, vs2, rs1, vm   # เลื่อนลงตามจำนวนองค์ประกอบใน  x[rs1]
vslidedown.vi vd, vs2, uimm, vm  # เลื่อนลงตามจำนวนองค์ประกอบใน  uimm
```

ตัวอย่าง เช่น หากความยาวเวกเตอร์ `vl` เป็น 64 และ `a0` มีค่า 16 ดังนั้น คำสั่ง `vslidedown.vx v0,v1,a0` จะคัดลอก 48 องค์ประกอบสุดท้ายของ `v1` ไปยัง 48 องค์ประกอบแรกของ `v0`

คำสั่ง `vslidedown` ช่วยในการลดขนาดข้อมูล โดยใช้แนวทางการแบ่งครึ่งแบบวนซ้ำ (recursive-halving approach) สำหรับตัวดำเนินการทวิภาคที่เป็นแบบเชิงสัมพันธ์

(binary associative operator) ตัวอย่างเช่น เพื่อคำนวณผลคูณขององค์ประกอบทั้งหมดในเรจิสเตอร์เวกเตอร์ ให้ใช้ `vslidedown` เพื่อคัดลอกครึ่งหลังของเวกเตอร์ไปยังครึ่งแรกของเรจิสเตอร์เวกเตอร์อื่น และลดความยาวเวกเตอร์ลงครึ่งหนึ่ง จากนั้น คูณเรจิสเตอร์เวกเตอร์ทั้งสองนี้เข้าด้วยกันและทำซ้ำการแบ่งครึ่งแบบวนซ้ำจนกว่าความยาวเวกเตอร์จะเท่ากับ 1 ผลลัพธ์ในองค์ประกอบที่ศูนย์ จะเป็นผลคูณขององค์ประกอบทั้งหมดในรีจิสเตอร์เวกเตอร์เดิม คำสั่ง `vslideup` มีลักษณะตรงข้ามกับ `vslidedown` โดยเลื่อนสมาชิกไปในทิศทางตรงกันข้าม

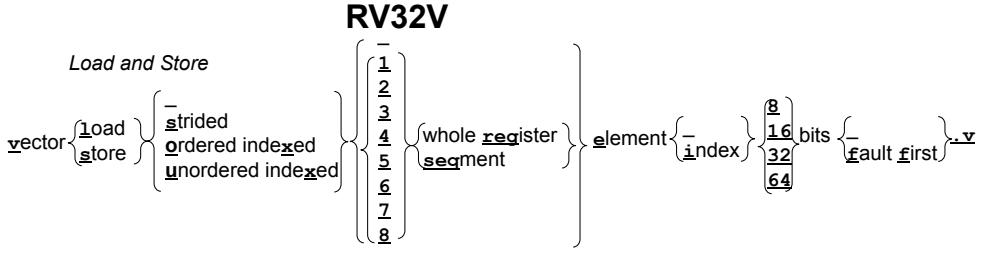
คำสั่งเลื่อนอื่นๆ อีกสี่คำสั่ง เลื่อนองค์ประกอบ เพียงหนึ่งตำแหน่ง แต่แทรกค่าแบบสเกลาร์ลงในช่องว่าง คำสั่ง `vslide1up` คัดลอกองค์ประกอบตั้งแต่ตำแหน่ง 0 ถึง `vl`-2 จากรีจิสเตอร์เวกเตอร์ต้นทางไปยังตำแหน่ง 1 ถึง `vl`-1 ในรีจิสเตอร์เวกเตอร์ปลายทาง จากนั้นวางรีจิสเตอร์ `x` แบบสเกลาร์ลงในตำแหน่ง 0 ของรีจิสเตอร์ปลายทาง คำสั่ง `vslide1down` คัดลอกองค์ประกอบตั้งแต่ตำแหน่ง 1 ถึง `vl`-1 จากรีจิสเตอร์เวกเตอร์ต้นทาง ไปยังตั้งแต่ตำแหน่ง 0 ถึง `vl`-2 ในรีจิสเตอร์เวกเตอร์ปลายทาง จากนั้นวางรีจิสเตอร์`x` แบบสเกลาร์ลงในตำแหน่ง `vl`-1 ของรีจิสเตอร์ปลายทาง `vfslide1up` และ `vfslide1down` ใช้รีจิสเตอร์ `f` แทนรีจิสเตอร์ `x`

```
vslide1up.vx    vd, vs2, rs1, vm   # vd[0] =   x[rs1],  vd[i+1] = vs2[i]
vfslide1up.vf   vd, vs2, rs1, vm   # vd[0] =   f[rs1],  vd[i+1] = vs2[i]
vslide1down.vx  vd, vs2, rs1, vm   # vd[i] = vs2[i+1], vd[vl-1] = x[rs1]
vfslide1down.vf vd, vs2, rs1, vm   # vd[i] = vs2[i+1], vd[vl-1] = f[rs1]
```

คำสั่งจัดการองค์ประกอบเวกเตอร์ตัวสุดท้าย `vcompress` จะบีบอัดองค์ประกอบที่ถูกเลือกโดยรีจิสเตอร์มาสก์เวกเตอร์ จากรีจิสเตอร์เวกเตอร์ต้นทางให้เรียงต่อกันที่ตำแหน่งเริ่มต้นของรีจิสเตอร์เวกเตอร์ปลายทาง:

```
vcompress.vm vd,vs2,vs1  # บีบอัดองค์ประกอบของ vs2
                         # ที่ vs1 ถูกเปิดใช้งานลงใน vd
```

การคำนวณมัลติเวิร์ด คำสั่งบวกจำนวนเต็มพร้อมแคร์รี่ และคำสั่งการลบแบบมียืม ช่วยเร่งการคำนวณเลขคณิตจำนวนเต็มแบบมัลติเวิร์ด รูปแบบแรก `vadc.vvm` และ `vsbc.vvm` ดำเนินการการบวกหรือการลบ ขนาด SEW บิต ตามปกติ แต่ยังบวกแบบแคร์รี่เข้า (carry-in) หรือลบแบบยืมเข้า (borrow-in) จากรีจิสเตอร์มาสก์ `v0` โดยให้ผลลัพธ์ขนาด SEW บิต รูปแบบที่สอง `vmadc.vvm` และ `vmsbc.vvm` ดำเนินการคำนวณแบบเดียวกัน แต่สร้างแคร์รี่ออก (carry-out) หรือยืมออก (borrow-out) ในรีจิสเตอร์มาสก์ รูปแบบสุดท้าย `vmadc.vv` และ `vmsbc.vv` คล้ายกันแต่ไม่ใช้แคร์รี่เข้าหรือยืมเข้า—ตัว `m` ถูกละไว้จากส่วนต่อท้าย—ทำให้มีประโยชน์สำหรับการสร้างแคร์รี่จากคำที่มีนัยสำคัญน้อยที่สุดในการคำนวณ



รูปที่ 8.8: แผนภาพของเวอร์ชันเต็มของคำสั่งถ่ายโอนข้อมูลซึ่งถูกละไว้จาก 8.1 เนื่องจากการพิมพ์แบบไดนามิก แผนภาพคำสั่งนี้ยังใช้งานได้โดยไม่ต้องเปลี่ยนแปลงสำหรับ RV64V ใน บทที่ 10

```
# สร้างผลบวกหรือผลลบแบบมีการทด
vadc.vvm vd, vs2, vs1, v0   # vd[i] = vs2[i] + vs1[i] + v0[i]
vsbc.vvm vd, vs2, vs1, v0   # vd[i] = vs2[i] - vs1[i] - v0[i]

# สร้างผลลัพธ์แบบมีการทด
vmadc.vvm vd, vs2, vs1, v0 # vd[i]=carry( vs2[i] + vs1[i] + v0[i] )
vmsbc.vvm vd, vs2, vs1, v0 # vd[i]=borrow( vs2[i] - vs1[i] - v0[i] )

# สร้างผลลัพธ์แบบไม่มีการทด
vmadc.vv vd, vs2, vs1      # vd[i]=carry( vs2[i] + vs1[i] )
vmsbc.vv vd, vs2, vs1      # vd[i]=borrow( vs2[i] - vs1[i] )
```

สำหรับการบวก RVV มีรูปแบบที่รับเรจิสเตอร์ `x` แบบสเกลาร์ (`.vx` และ `.vxm`) หรือค่าคงที่ที่ขยายเครื่องหมาย (`.vi` และ `.vim`) สำหรับการลบ มีรูปแบบที่ใช้เรจิสเตอร์ `x` แต่ไม่มีรูปแบบที่ใช้ค่าคงที่

การถ่ายโอนข้อมูล หมวดหมู่สุดท้ายของคำสั่งไลบรารีช่วยในการสลับบริบท ป้องกันข้อยกเว้นที่ไม่ตั้งใจในการถ่ายโอนข้อมูลเวกเตอร์ และการจัดการอาร์เรย์ของโครงสร้าง แผนภาพคำสั่งของคำสั่งเหล่านี้อยู่ใน รูปที่ 8.8

โหลดและบันทึกรีจิสเตอร์เวกเตอร์ทั้งหมด ช่วยบันทึกและกู้คืนเรจิสเตอร์เวกเตอร์เมื่อไม่ทราบประเภทหรือความยาวของเนื้อหาปัจจุบันในเรจิสเตอร์เวกเตอร์ หรือเมื่อการแก้ไข `vl` และ `vtype` จะมีค่าใช้จ่ายสูง ตัวอย่างรวมถึงการรั่วไหลของเรจิสเตอร์ของคอมไพเลอร์ (compiler register spills) การเรียกฟังก์ชันเวกเตอร์ที่ส่งค่าในเรจิสเตอร์เวกเตอร์ ตัวจัดการการขัดจังหวะ และการสลับบริบทของระบบปฏิบัติการ ตัวอย่างด้านล่างนี้ใช้สำหรับคำ 32 บิต แต่ยังมีรุ่นของคำสั่งสำหรับไบต์ ฮาล์ฟเวิร์ด (halfwords) และดับเบิลเวิร์ด (doublewords)

```
vl1re32.v v3, (a0)  # โหลด v3     ด้วย 1*VLEN/32 เวิร์ดจากที่อยู่ a0
vl2re32.v v2, (a0)  # โหลด v2-v3  ด้วย 2*VLEN/32 เวิร์ดจากที่อยู่ a0
vl4re32.v v4, (a0)  # โหลด v4-v7  ด้วย 4*VLEN/32 เวิร์ดจากที่อยู่ a0
vl8re32.v v8, (a0)  # โหลด v8-v15 ด้วย 8*VLEN/32 เวิร์ดจากที่อยู่ a0
vs1r.v    v3, (a0)  # บันทึก v3     ไปยังที่อยู่ a0
vs2r.v    v2, (a0)  # บันทึก v2-v3  ไปยังที่อยู่ a0
vs4r.v    v4, (a0)  # บันทึก v4-v7  ไปยังที่อยู่ a0
vs8r.v    v8, (a0)  # บันทึก v8-v15 ไปยังที่อยู่ a0
```

คำสั่ง `vl1r` เข้ารหัสความกว้างขององค์ประกอบ แม้ว่าจะใช้การใช้งาน `vl1re8` ตลอดเวลาะถูกต้องตามฟังก์ชันการทำงาน แต่คำสั่งอื่นๆ ให้คำใบ้แก่การใช้งาน เกี่ยวกับวิธีที่ผลลัพธ์ของการโหลดจะถูกนำไปใช้ ซึ่งสามารถช่วยปรับปรุงการใช้งานที่ปรับแต่งเค้าโครงของไฟล์ (register-file layout) แบบไดนามิกสำหรับความกว้างขององค์ประกอบที่แตกต่างกัน ข้อพิจารณาเดียวกันนี้ไม่ใช้กับคำสั่งบันทึก `vs1r` เพราะการใช้งานทราบประเภทข้อมูลที่ใช้ล่าสุดอยู่แล้ว ดังนั้นคำสั่ง `vs1r` จึงไม่จำเป็นต้องเข้ารหัสความกว้างขององค์ประกอบ

การโหลดแบบก้าวหน่วยที่เกิดข้อผิดพลาดเฉพาะตัวแรก (Unit-stride Fault-Only-First Loads) ช่วยให้สามารถทำให้ลูปเป็นเวกเตอร์ได้สำหรับลูปที่ไม่ทราบจำนวนรอบ (trip count) เมื่อเริ่มลูป เช่น การคำนวณความยาวของสตริงที่สิ้นสุดด้วยค่า null ความท้าทายในการทำให้ลูปเช่นนี้เป็นเวกเตอร์คือ การโหลดเวกเตอร์อาจเผลอเข้าถึงเกินขอบเขตของโครงสร้างข้อมูล ซึ่งจะทำให้เกิดข้อยกเว้น (exception) การโหลดแบบ fault-only-first ช่วยแก้ปัญหานี้ได้ โดยมันทำงานเหมือนการโหลดแบบก้าวหน่วย (unit-stride loads) ทั่วไป ยกเว้นว่ามันจะเกิดการดักจับ (trap) เฉพาะเมื่อมีข้อยกเว้นที่องค์ประกอบที่ 0 เท่านั้น หากตรวจพบข้อยกเว้นที่องค์ประกอบ 0 การดักจับจะเกิดขึ้นและ `vl` จะไม่ถูกแก้ไข หากตรวจพบข้อยกเว้นที่องค์ประกอบถัดไป การดักจับที่เกี่ยวข้องจะ*ไม่*เกิดขึ้น แทนที่นั้น ความยาวเวกเตอร์ `vl` จะถูกลดลงเป็นดัชนีขององค์ประกอบที่ควรจะทำให้เกิดข้อยกเว้น วิธีนี้ให้โอกาสซอฟต์แวร์ในการพิจารณาว่าควรจะไม่ดักจับข้อยกเว้นนั้นหรือไม่ (เช่น เพราะพบตัวสิ้นสุด null ในองค์ประกอบก่อนหน้า) มิฉะนั้น ในการวนลูปครั้งถัดไป ข้อยกเว้นเดียวกันจะถูกตรวจพบที่องค์ประกอบที่ 0 ซึ่งในจุดนั้นการดักจับจะเกิดขึ้น ต่อไปนี้คือคำสั่ง:

```
# vd ปลายทาง rs1 ที่อยู่ฐาน vm มาสก์
vle8ff.v  vd, (rs1), vm    # การโหลดแบบก้าวหน่วย 8 บิต เฉพาะข้อยกเว้นแรก
vle16ff.v vd, (rs1), vm    # การโหลดแบบก้าวหน่วย 16 บิต เฉพาะข้อยกเว้นแรก
vle32ff.v vd, (rs1), vm    # การโหลดแบบก้าวหน่วย 32 บิต เฉพาะข้อยกเว้นแรก
vle64ff.v vd, (rs1), vm    # การโหลดแบบก้าวหน่วย 64 บิต เฉพาะข้อยกเว้นแรก
```

ชุดสุดท้ายของการถ่ายโอนข้อมูล *การโหลดและบันทึกเซกเมนต์เวกเตอร์* ช่วยในการทำให้อาร์เรย์ของโครงสร้างเป็นเวกเตอร์ แต่ละฟิลด์ของโครงสร้างสามารถโหลดลงในเรจิสเตอร์เวกเตอร์แยกกันได้ โดยคำสั่งเหล่านี้รองรับตั้งแต่ 1 ถึง 8 ฟิลด์ เราจะแสดงการทำงานของมันด้วยตัวอย่างสองตัวอย่าง พิจารณาโครงสร้างที่เก็บพิกเซล RGB (แดง, เขียว, น้ำเงิน) ที่ถูกบีบอัด โดยแต่ละสีมีขนาด 8 บิตในโครงสร้างข้อมูล 24 บิต โดยการตั้งค่าจำนวนฟิลด์เป็น 3 และเริ่มต้นด้วยเรจิสเตอร์เวกเตอร์ v8 เราจะโหลดพิกเซลสีแดงลงใน v8, พิกเซลสีเขียวลงใน v9, และพิกเซลสีน้ำเงินลงใน v10:

```
vlseg3e8.v v8, (a0)
```

การใช้งานยอดนิยมอีกอย่างหนึ่งคือจำนวนเชิงซ้อน ตัวอย่างนี้ใช้ `vlseg` เพื่อแยกโครงสร้างของจำนวนเชิงซ้อนออกเป็นส่วนจริง 32 บิตและส่วนจินตภาพ 32 บิต `v8` และ

v9 จากนั้นสร้างคอนจูเกตเชิงซ้อน (complex conjugates) โดยการทำให้ส่วนจินตภาพเป็นลบ แล้วบีบอัดผลลัพธ์กลับเข้าไปในโครงสร้างในหน่วยความจำโดยใช้ vsseg

```
vlseg2e32.v v8, (a0)    # โหลดส่วนจริงลงใน v8 ส่วนเชิงซ้อนลงใน v9
vfsgnjn.vv v9, v9, v9   # ทำให้ส่วนจินตภาพเป็นลบ
vsseg2e32.v v8, (a0)    # บันทึกลงในหน่วยความจำในรูปแบบบีบอัด
```

นอกเหนือแบบก้าวหน่วย (unit stride) ของการโหลดและบันทึกเซกเมนต์แล้ว RVV ยังมีแบบก้าวข้าม (strided) และแบบใช้ดัชนี (indexed) ของเซกเมนต์ด้วย แบบหลังมีทั้งแบบเรียงลำดับ และไม่เรียงลำดับ ตัวอย่างเช่น:

```
vlsseg3e8.v v4, (x5), x6     # โหลดไบต์จาก (x5 + i*x6) ลงใน v4[i]
                             # และไบต์จาก (x5 + i*x6 + 1) ลงใน v5[i]
                             # และไบต์จาก (x5 + i*x6 + 2) ลงใน v6[i]
vsuxseg2ei32.v v7, (x5), v3  # บันทึกแบบไม่เรียงลำดับโดยใช้
                             # องค์ประกอบขนาด SEW บิต และดัชนี 32 บิต
                             #  v7[i] บันทึกไปยัง (x5 + v3[i])
                             #  v8[i] บันทึกไปยัง (x5 + v3[i] + (SEW/8))
```

ในขั้นตอนสุดท้ายของการเจาะลึก รูปที่ 8.13 ถึง 8.17 ที่ส่วนท้ายบทแสดงรูปแบบคำสั่ง และการเข้ารหัสของคำสั่งเวกเตอร์ที่นิยมในรูปที่ 8.1

8.12 การเปรียบเทียบ RV64V กับ ARMv8 SVE และ x86-64 AVX SIMD

เราจะเปรียบเทียบการดำเนินการ DAXPY บนสถาปัตยกรรมเวกเตอร์ RISC-V กับ x86 Advanced Vector eXtension (AVX) ที่ SIMD 128 บิต และ 512 บิต และจากนั้นจึงเปรียบเทียบกับการดำเนินการบน ARMv8 Scalable Vector Extension (SVE)

ตัว (pd ใน vfmadd213pd ที่ตำแหน่ง 15 หมายถึงการทำการคูณ-บวกแบบขนาน 4 ครั้ง ในขณะที่ sd ใน vfmadd213sd ที่ตำแหน่ง 33 จะทำการคูณ-บวกเพียงครั้งเดียว ค่าใช้จ่ายในการจัดการบัญชี (bookkeeping overhead) ของ SIMD ที่ตำแหน่ง a ถึง e และ 29 ถึง 44 เห็นได้ชัดเมื่อเปรียบเทียบโค้ดนี้กับโค้ด RVV ใน รูปที่ 8.4 โค้ดการจัดการบัญชีเช่นนี้ไม่จำเป็นสำหรับ RVV เพราะเรจิสเตอร์ความยาวเวกเตอร์ vl และคำสั่ง vsetvl ทำให้ลูปทำงานได้กับค่าใดๆ ของ n

x86-64 AVX2 SIMD Intel ได้ผ่านการพัฒนาหลายรุ่นของส่วนขยาย SIMD ซึ่งเราเห็นได้ในโค้ดใน รูปที่ 8.9 การขยาย Streaming SIMD Extensions (SSE) ไปสู่ SIMD ขนาด 128 บิตนำไปสู่การมีรีจิสเตอร์ xmm 8 ตัว และคำสั่งที่สามารถใช้งานได้ การขยายไปสู่ SIMD 256 บิต ซึ่งในส่วนของ Advanced Vector Extensions (AVX) สร้างรีจิสเตอร์ ymm 16 ตัว และคำสั่งใหม่จำนวนมากที่สามารถใช้งานได้

กลุ่มคำสั่งแรกที่ตำแหน่งที่อยู่ 0 ถึง e ในรูปที่ 8.9 โหลดตัวแปรจากหน่วยความจำ ทำสำเนา a สี่ชุดในรีจิสเตอร์ ymm ขนาด 128 บิต และทดสอบเพื่อให้แน่ใจว่า n มีค่าอย่างน้อย 4 ก่อนเข้าสู่ลูปหลัก ส่วนนำ (preamble) นี้ใช้คำสั่ง AVX หนึ่งคำสั่ง (คำบรรยายของรูปที่ 8.9 อธิบายรายละเอียดเพิ่มเติม)

ลูปหลักดำเนินการคำนวณส่วนสำคัญของ DAXPY คำสั่ง AVX vmovupd ตำแหน่งที่อยู่ 10 โหลดสี่องค์ประกอบของ x ลงใน ymm0 คำสั่ง AVX vfmadd213pd ที่ตำแหน่งที่อยู่

```
# rdi คือ n, rsi ตัวชี้ไปที่ x[0], rdx ตัวชี้ไปที่ y[0], xmm0 คือ a
  0: c4 e2 7d 19 d0    vbroadcastsd ymm2, xmm0  # ymm2 = 4 copies of a
  5: 31 c0             xor    eax, eax          # i = 0
  7: 48 89 f9          mov    rcx, rdi          #
  a: 48 83 e1 fc       and    rcx, -4           # rcx = floor(n / 4) * 4
  e: 74 19             je after_avx_loop        # skip main loop if n < 4
avx_loop:
 10: c5 fd 10 04 c6    vmovupd ymm0, [rsi+rax*8]  # load x
 15: c4 e2 ed a8 04 c2 vfmadd213pd ymm0, ymm2, [rdx+rax*8] # 4 mul-adds
 1b: c5 fd 11 04 c2    vmovupd [rdx+rax*8], ymm0  # store y
 20: 48 83 c0 04       add    rax, 4             # i += 4
 24: 48 39 c1          cmp    rcx, rax           # i < floor(n / 4) * 4?
 27: 75 e7             jne avx_loop              # Repeat if so
after_avx_loop:
 29: 48 39 c7          cmp    rdi, rax           # Any fringe elements?
 2c: 76 18             jbe after_scalar_loop     # Skip if not
scalar_loop:
 2e: c5 fb 10 04 c6    vmovsd xmm0, [rsi+rax*8]   # load x
 33: c4 e2 e9 a9 04 c2 vfmadd213sd xmm0, xmm2, [rdx+rax*8] # 1 mul-add
 39: c5 fb 11 04 c2    vmovsd [rdx+rax*8], xmm0   # store y
 3e: 48 ff c0          inc    rax                # i++
 41: 48 39 c7          cmp    rdi, rax           # i < n?
 44: 75 e8             jne  scalar_loop          # Repeat if so
after_scalar_loop:
 46: c3                ret                       # Return
```

รูปที่ 8.9: โค้ด x86-64 AVX2 สำหรับ DAXPY ในรูปที่ 5.5 คำสั่ง AVX `vbroadcastsd` ที่ตำแหน่งที่อยู่ 0 สร้างสำเนาของ `a` สี่ชุดใน `ymm2` การทดสอบสำหรับ `n` < 4 ที่ตำแหน่ง e ตรวจสอบว่าควรข้ามลูปหลักหรือไม่ คำสั่ง AVX สามคำสั่งที่ตำแหน่งที่อยู่ 2e ถึง 39 (`vmovsd`, `vfmadd213sd`, `vmovsd`) จัดการเมื่อ mod(`n`,4) ≠ 0 โดยดำเนินการคำนวณ DAXPY ทีละองค์ประกอบ โดยลูปจะทำซ้ำจนกว่าฟังก์ชันจะดำเนินการการคูณ-บวกครบตามจำนวน mod(`n`,4) ครั้ง

```
# rdi คือ n, rsi ตัวชี้ไปที่ x[0], rdx ตัวชี้ไปที่ y[0], xmm0 คือ a
  0: 89 f9                mov     ecx, edi     #
  2: 83 e1 07             and     ecx, 7       # Form mask for fringe
  5: 41 ba 01 00 00 00    mov     r10d, 1      #   cases by computing
  b: 41 d3 e2             shl     r10d, cl     #   2^(n mod 8) - 1
  e: 41 ff ca             dec     r10d         #
 11: 62 f2 fd 48 19 d0    vbroadcastsd zmm2, xmm0  # zmm2 = 8 copies of a
 17: 48 89 f9             mov     rcx, rdi     #
 1a: 48 83 e1 f8          and     rcx, -8      # rcx = floor(n / 8) * 8
 1e: c5 f5 46 c9          kxnorb k1, k1, k1    # mask = all 1s
 22: 31 c0                xor     eax, eax     # i = 0
 24: eb 1f                jmp     tail         # กระโดดไปที่ loop tail
loop:
 26: 62 f1 fd c9 10 04 c6 vmovupd zmm0{k1}{z}, [rsi+rax*8] # load x
 2d: 62 f1 fd c9 10 0c c2 vmovupd zmm1{k1}{z}, [rdx+rax*8] # load y
 34: 62 f2 f5 c9 98 c2    vfmadd132pd zmm0{k1}{z}, zmm1, zmm2 # 8 mul-adds
 3a: 62 f1 fd 49 11 04 c2 vmovupd [rdx+rax*8]{k1}, zmm0    # store y
 41: 48 83 c0 08          add     rax, 8                   # i += 8
tail:
 45: 48 39 c1             cmp     rcx, rax  # i < floor(n / 8) * 8?
 48: 77 dc                ja      loop      # Repeat if so
 4a: c4 c1 7b 92 ca       kmovd   k1, r10d  # Install mask for fringe cases
 4f: 48 39 c7             cmp     rdi, rax  # i < n?
 52: 77 d2                ja      loop      # Perform final iteration if so
 54: c3                   ret               # Return
```

รูปที่ 8.10: โค้ด x86-64 AVX-512 สำหรับ DAXPY ใน รูปที่ 5.5 ลูปหลักที่ตำแหน่ง 26 ถึง 41 คล้ายกับรูปที่ 8.9 มาก เพียงว่าโค้ดนี้ใช้การโหลดสองครั้ง และการคูณ-บวก ในขณะที่โค้ดก่อนหน้าใช้การโหลดครั้งเดียว และมีตัวถูกดำเนินการหน่วยความจำในคำสั่งคูณ-บวก ความแตกต่างใหญ่คือการใช้มาสก์สำหรับการขุดแอบ (stripmining) คำสั่งที่ตำแหน่ง 2 ถึง e เริ่มต้นมาสก์ และคำสั่งที่ตำแหน่ง 45 ถึง 52 ปรับหน้ากากสำหรับเมื่อ mod(n,8) ≠ 0 การดำเนินการมีมาสก์คำนวณองค์ประกอบที่เหลือทั้งหมดในขั้นตอนเดียว แทนที่จะคำนวณทีละองค์ประกอบอย่างในรูปที่ 8.9

15 คูณสำเนาสี่ชุดของ a (ymm2) กับสี่องค์ประกอบของ x (ymm0) บวกกับสี่องค์ประกอบของ y (ในหน่วยความจำตำแหน่งที่อยู่ ecx+edx*8) และใส่ผลรวมสี่ตัวลงใน ymm0 คำสั่ง AVX ถัดไปนี้ที่ตำแหน่งที่อยู่ 1b, vmovupd บันทึกผลลัพธ์สี่ตัวลงใน y คำสั่งสามคำสั่งถัดไปเพิ่มตัวนับและทำซ้ำลูปหากจำเป็น

ต่างจาก RVV เนื่องจากไม่มีรีจิสเตอร์ความยาวเวกเตอร์ AVX จึงต้องมีคำสั่งจัดการบัญชีเพิ่มเติมเพื่อตรวจสอบค่าที่มีปัญหาของ n โค้ด "ขอบ" (fringe) ระหว่างที่อยู่ 29 และ 44 จัดการกรณีเมื่อ n ไม่ใช่ทวีคูณของ 4 โดยใช้คำสั่ง AVX สามคำสั่ง

คำสั่ง 6 คำสั่งของลูปหลักในโค้ด DAXPY ของ x86-64 AVX2 ทำการเข้าถึงหน่วยความจำความแบบแม่นยำสองเท่า 12 ครั้ง และการดำเนินการจุดทศนิยม 8 ครั้ง โดยเฉลี่ย 2 การเข้าถึงหน่วยความจำและประมาณ 1.3 การดำเนินการต่อคำสั่ง

โค้ด การ จัดการ บัญชี (bookkeeping code) ถือ เป็น ส่วน หนึ่ง ของ stripmining ตาม คำ อธิบายของรูปที่ 8.9 อธิบาย รีจิสเตอร์ความยาวเวกเตอร์ vl ทำให้โค้ด การ จัดการ บัญชี SIMD ไม่ จำเป็น สำหรับ RVV สถาปัตยกรรม แบบ เวก เตอร์ แบบ ดั้งเดิม ต้องการ โค้ด เพิ่ม เติม เพื่อ จัดการ กรณี มุม (corner case) ของ n = 0 แต่ RVV เพียงทำให้คำสั่งเวกเตอร์ทำหน้าที่เหมือนคำสั่ง nops เมื่อ vl = 0

x86-64 AVX-512 SIMD ขั้นตอนต่อไปในวิวัฒนาการของ x86 เป็น 512-บิต SIMD เรียกว่า AVX-512 นอกเหนือจากคำสั่ง SIMD 512 บิตใหม่ มันยังขยายเป็น 32 zmm รีจิส

```
# x0 คือ n, x1 ตัวชี้ไปที่ x[0], x2 ตัวชี้ไปที่ y[0], d0 คือ a
   0: d2800003  mov  x3, 0                          # i = 0
   4: 05282000  mov  z0.d, d0                       # broadcast a across vector z0
loop:
   8: 25e01c60  whilelo p0.d, x3, x0                # form predicates from i, n
   c: a5e34021  ld1d z1.d, p0/z, [x1, x3, lsl 3] # load vector x
  10: a5e34042  ld1d z2.d, p0/z, [x2, x3, lsl 3] # load vector y
  14: 65e28001  fmad z1.d, p0/m, z0.d, z2.d       # z1 = z1 * z0 + z2
  18: e5e34041  st1d z1.d, p0, [x2, x3, lsl 3]    # store vector y
  1c: 04f0e3e3  incd x3                           # i += number of doubles in a vector
  20: eb03001f  cmp x0, x3                        # n > i?
  24: 54ffff28  b.hi loop                         # branch if so
  28: d65f03c0  ret                               # return
```

รูปที่ 8.11: โค้ด ARMv8 SVE สำหรับ DAXPY ในรูปที่ 5.5 เปรียบเทียบโค้ดนี้กับ รูปที่ 8.4 ทั้งคู่ใช้ 11 คำสั่ง แม้ว่า SVE จะมีคำสั่งในลูปน้อยกว่า (ดูรูปที่ 8.12) การแบ่งแยกส่วนดำเนินการโดยใช้มาสก์ เช่นเดียวกับ AVX-512 ในรูปที่ 8.10

เตอร์ และเพิ่ม รีจิสเตอร์มาสก์ ซึ่งเป็นส่วนประกอบมาตรฐาน ของสถาปัตยกรรมเวกเตอร์แบบดั้งเดิม นอกจากจะช่วยในการใช้คำสั่ง SIMD เมื่อลูปมีคำสั่งตามเงื่อนไขแล้ว มาสก์เหล่านี้ยังลดค่าใช้จ่ายการแยกส่วนของ DAXPY ถ้า `n` ไม่ใช่ทวีคูณของ 8 โค้ดที่ตำแหน่ง 4a จะตั้งค่ามาสก์แล้วใช้มาสก์นั้นเพื่อดำเนินการวนลูปครั้งสุดท้ายหนึ่งรอบ ซึ่งมีระหว่าง 1 ถึง 7 การดำเนินการ ขึ้นอยู่กับค่าของ mod(`n`,8)

รูปที่ 8.10 หน้าที่ 109 แสดงคำสั่ง AVX-512 ของ DAXPY คำสั่ง 7 คำสั่งของลูปหลักในโค้ด x86-64 AVX-512 DAXPY ทำการเข้าถึงหน่วยความจำความแม่นยำสองเท่า 24 ครั้ง และการดำเนินการจุดทศนิยม 16 ครั้ง โดยเฉลี่ย 3.4 การเข้าถึงหน่วยความจำและ 2.3 การดำเนินการต่อคำสั่ง

ARM64 SVE Vector รูปที่ 8.11 หน้าที่ 110 แสดงคำสั่ง ARMv8 SVE ของ DAXPY เนื่องจากนี่เป็นการกล่าวถึงคำสั่ง ARMv8 ครั้งแรกของเรา เราจะอธิบายทั้ง 11 คำสั่ง

1. คำสั่ง `mov` ที่ตำแหน่ง 0 เริ่มต้นด้วยการกำหนดค่าเริ่มต้นให้ตัวแปร `i` (`x3`) เป็น 0
2. คำสั่ง `mov` ถัดไปจะคัดลอกค่าในตัวแปร `a` ซึ่งแสดงด้วย รีจิสเตอร์จุดทศนิยมสเกลาร์ความแม่นยำสองเท่า `d0` ไปยังทุกองค์ประกอบของรีจิสเตอร์เวกเตอร์ความแม่นยำสองเท่า `z0.d` ซึ่งใช้ในขั้นตอนที่ 6
3. คำสั่งที่ตำแหน่ง 8 เป็นคำสั่งแรกของลูป ที่มีบทบาทเดียวกับคำสั่ง `vsetvli` ในโค้ด RVV DAXPY ในรูปที่ 8.4 โดยใช้มาสก์เหมือน AVX-512 แทนการปรับ `vl` คำสั่ง `whilelo` ตั้งค่าบิตมาสก์ `p0/z` เป็น 1 สำหรับทุกองค์ประกอบที่ `i` (`x3`) ต่ำกว่า `n` (`x0`) ซึ่งให้ผลตามที่ต้องการ
4. การโหลดเวกเตอร์ `x` ของ DAXPY เป็นขั้นตอนต่อไป คำสั่ง `LD1D z1.d` เป็นการโหลดแบบก้าวหน่วยของดับเบิลเวิร์ดไปยังรีจิสเตอร์เวกเตอร์ดับเบิลเวิร์ด `z1.d` ฟิลด์ถัดไป `p0/z` หมายถึงมันทำงานภายใต้มาสก์ `p0` ที่กำหนดโดย `whilelo` ส่วนตำแหน่งที่อยู่ `[x1, x3, lsl 3]` หมายถึงที่อยู่ฐานใน `x1` (ที่อยู่ของเวกเตอร์ `x`)

ถูกบวกกับตัวแปร `i` คูณด้วย 8 (โดยการเลื่อนซ้าย 3 บิต) เพื่อเปลี่ยนดัชนีดับเบิลเวิร์ดเป็นออฟเซ็ตไบต์ การเลื่อนนี้เป็นคำสั่งแยกต่างหากที่ตำแหน่ง 16 ในโค้ด RVV DAXPY ใน รูปที่ 8.4

5. การโหลดที่ตำแหน่ง 10 ดำเนินการโหลดแบบเดียวกัน สำหรับเวกเตอร์ `y` ของ DAXPY ลงในรีจิสเตอร์เวกเตอร์ดับเบิลเวิร์ด `z2.d`
6. คำสั่ง `fmad z1.d, p0/m, z0.d, z2.d` คูณ `z1.d` (`x`) ด้วย `z0.d` (`a` ที่ตั้งในขั้นตอนที่ 2) บวกกับ `z2.d` (`y`) และใส่ผลรวมกลับลงใน `z1.d` การดำเนินการนี้ทำภายใต้มาสก์ `p0` เดียวกัน ARM SVE ไม่มีคำสั่งคูณ-บวกแบบเวกเตอร์-สเกลาร์ ซึ่งเป็นเหตุผลที่โค้ด SVE ต้องการคำสั่งในขั้นตอนที่ 2 เพื่อทำซ้ำ `a` ในรีจิสเตอร์เวกเตอร์ แต่ RVV ไม่ต้องการ
7. คำสั่งบันทึก `st1d` ที่ตำแหน่ง 18 คล้ายกับการโหลดในขั้นตอนที่ 5 เพราะมันบันทึกผลลัพธ์ลงในที่อยู่เดียวกันของเวกเตอร์ `y` แทนการโหลด
8. คำสั่ง `incd x3` เพิ่มค่า `i` (`x3`) ด้วยจำนวนดับเบิลเวิร์ดในรีจิสเตอร์เวกเตอร์โปรเซสเซอร์คำนวณจำนวนนั้นจากความยาวของรีจิสเตอร์เวกเตอร์ RVV แทนที่จะลด `i` ด้วยความยาวของเวกเตอร์โดยใช้คำสั่งสเกลาร์ `sub` มาตรฐาน
9. คำสั่ง `cmp` ที่ตำแหน่ง 20 เปรียบเทียบ `n` (`x0`) กับ `i` (`x3`) และตั้งรหัสเงื่อนไข
10. คำสั่งกระโดดตามเงื่อนไข `b.hi` กระโดดไปที่จุดเริ่มต้นของลูปหาก `n > i`
11. หากคำสั่งกระโดดแบบมีเงื่อนไขไม่ถูกดำเนินการ คำสั่ง `ret` จะกลับไปยังคืนการควบคุมไปยังผู้เรียก (callee)

คุณอาจสงสัยว่าโค้ด SVE จะมีปัญหาเมื่อ `n` เป็น 0 แต่ไม่มี เพราะเมื่อ `n` = 0 SVE จะตั้งบิตมาสก์ทั้งหมดเป็น 0 ดังนั้นจึงไม่มีอะไรเปลี่ยนแปลง เช่นเดียวกัน คำสั่งเวกเตอร์ RVV จะไม่เปลี่ยนแปลงเมื่อ `vl` = 0

การอภิปราย รูปที่ 8.12 สรุปจำนวนคำสั่งและจำนวนไบต์ใน DAXPY ของโปรแกรมสำหรับ x86-64 AVX2, x86-64 AVX-512, ARMv8 SVE และ RV64GCV

โค้ดการคำนวณ SIMD ถูกบดบังด้วยโค้ดการจัดการบัญชี ส่วนใหญ่ของโค้ดสำหรับ SIMD เป็นค่าใช้จ่าย —70% สำหรับ AVX2 และ 68% สำหรับ AVX-512 —ไม่ว่าจะเป็นการเตรียมข้อมูลสำหรับลูป SIMD หลัก หรือสำหรับการขุดแถบ SIMD โค้ด RVV ในรูปที่ 8.4 และโค้ด SVE ใน รูปที่ 8.11 ไม่ต้องการโค้ดการจัดการบัญชีเช่นนี้ ต่างจาก SIMD ทั้ง RVV และ SVE มีกลไกควบคุมความยาวเวกเตอร์ (`vl` สำหรับ RVV และ `whilelo` สำหรับ SVE) ซึ่งทำให้คำสั่งเวกเตอร์ทำงานได้กับทุกค่า `n` ใดๆ โค้ดของสถาปัตยกรรมเวกเตอร์ทั้งสองมีขนาดเล็กกว่ารุ่น x86 ขนาดโค้ด RVV เล็กกว่า SVE เพราะ RVV สามารถรวมกับคำสั่งบีบอัดของ RVC ได้ ในขณะที่ ARMv8 นั้นกำหนดให้ทุกคำสั่งกว้าง 32 บิต โค้ด RV64GCV เล็กกว่า SVE 1.4 เท่า เล็กกว่า AVX2 2.2 เท่า และเล็กกว่า AVX-512 2.6 เท่า แม้จะมีคำสั่งขนาดตัวแปรไบต์ แนวทางการเพิ่มขั้นของ ISA ส่งผลเสียต่อขนาดคำสั่ง:

ในขณะที่ ISA ARM-32 รุ่นก่อนหน้ามี Thumb และ Thumb2 เพื่อบีบอัดขนาดโค้ด สถาปนิกของ ARMv8 ละทิ้งการบีบอัดโค้ดเพื่อลดความซับซ้อนของการใช้งานแบบ superscalar การเพิ่มคำสั่งบีบอัดย้อนหลังให้กับ ISA เป็นเรื่องยาก เพราะพื้นที่รหัสดำเนินการที่มีค่าที่จำเป็นสำหรับคำสั่งขนาดเล็กถูกครอบครองไปแล้ว



ความเรียบง่าย

ISA	x86-64 AVX2	x86-64 AVX-512	ARMv8 SVE	RV64GCV
Instructions (static)	20	22	11	11
Bytes (static)	70	84	44	32
Instructions per Main Loop	6	7	8	10
Results per Main Loop (512B state)	4	–	2	16
Instructions (dynamic, n=1000)	1508	–	4003	631
Results per Main Loop (2048B state)	–	8	8	64
Instructions (dynamic, n=1000)	–	892	1003	161

รูปที่ 8.12: จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับ SIMD และ ISA แบบเวกเตอร์ ตารางนี้ระบุจำนวนคำสั่งทั้งหมด (แบบคงที่), ขนาดโค้ด จำนวนคำสั่ง แถวที่ 5 และ 6 แสดงผลลัพธ์ต่อลูปและจำนวนคำสั่งที่ดำเนินการโดยสมมติว่า `n` = 1000 และสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์มีขนาด 512 ไบต์ ซึ่งเท่ากับขนาดสถานะรีจิสเตอร์ SIMD ของ AVX2 แถวที่ 7 และ 8 แสดงตัวชี้วัดเดียวกันแต่สถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์มีขนาด 2048 ไบต์ เท่ากับขนาดสถานะรีจิสเตอร์ SIMD ของ AVX-512 SVE ดำเนินการคำสั่งมากที่สุดสำหรับแต่ละขนาดสถานะ และ LMUL ทำให้ RV64V ดำเนินการคำสั่งน้อยที่สุด โปรดทราบว่าโค้ดเวกเตอร์ทั้งหมดไม่เปลี่ยนแปลง ไม่ว่าความยาวเวกเตอร์จะเป็นเท่าใด แต่ต้องคอมไพล์ใหม่เพื่อเพิ่มความกว้าง SIMD จาก 128 เป็น 512 บิต เนื่องจากใช้คำสั่งที่แตกต่างกัน

คำสั่ง AVX2 ยาว 6 ไบต์ และ 7 ไบต์สำหรับ AVX-512 ใน DAXPY ความยาวคำสั่งเฉลี่ยเพิ่มจาก 3.5 สำหรับ AVX2 เป็น 3.8 สำหรับ AVX-512 เทียบกับ 2.9 ไบต์สำหรับ RV64CV และ 4.0 สำหรับ SVE

นอกเหนือจากโค้ดภาษาเครื่องที่เล็กลง โค้ดเวกเตอร์ยังเข้าใจง่ายกว่า SIMD มาก เมื่อเปรียบเทียบรุ่นสเกลาร์และเวกเตอร์ของ RISC-VDAXPY ในจากรูปที่ 5.7 หน้า 65 และรูปที่ 8.4 หน้า 94 ทั้งคู่มีจำนวนคำสั่งเท่ากัน ประกอบด้วย 2 การโหลด, 1 การคูณ-บวก, 1 การเลื่อน, 1 การบันทึก, และ 1 การคืนค่า แบบแรก มี 3 คำสั่งบวกและคำสั่งกระโดดแบบมีเงื่อนไข 2 คำสั่ง และแบบหลัง มี 2 คำสั่งบวก,1 คำสั่งลบ, คำสั่งกระโดดมีเงื่อนไข 1 คำสั่งและ 1 คำสั่ง `vsetvli` รุ่น SIMD ของ x86 มีคำสั่งมากกว่ารุ่นสเกลาร์สองเท่า และยากที่จะเข้าใจว่าโค้ดทำอะไรหากไม่บอกล่วงหน้า


ความเรียบง่าย


ประสิทธิภาพ

อย่างไรก็ตาม ความแตกต่างที่สำคัญที่สุดระหว่าง SIMD และการประมวลผลแบบเวกเตอร์ไม่ใช่ขนาดโค้ดแบบคงที่ แม้ว่า RVV และ AVX ใช้ขนาดสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์เท่ากันที่ 512 ไบต์ SIMD ดำเนินการคำสั่งมากกว่า RVV 2.4 เท่า เพราะแต่ละลูป SIMD คำนวณเพียง 4 องค์ประกอบ แทนที่จะเป็น 16 สำหรับ RVV การดึงคำสั่งและถอดรหัสคำสั่งเพิ่มเติมหมายถึง พลังงานที่สูงขึ้นในการทำงานเดียวกัน

แถวถัดไป 2 สองของรูปที่ 8.12 แสดงข้อดีอีกประการของสถาปัตยกรรมแบบเวกเตอร์ โค้ดเดียวกันทำงานได้โดยไม่เปลี่ยนแปลงบนสถาปัตยกรรมแบบเวกเตอร์ที่มีสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์ 2048 ไบต์ โดยการเพิ่มสถานะ 4 เท่าลดจำนวนคำสั่งลงเกือบ 4 เท่า เมื่อเทียบกับ AVX-512 RVV ดำเนินการคำสั่งน้อยลงเกือบ 5.5 เท่า

หาก ค่า ใช้ จ่าย การดำเนินการสเกลาร์จำกัดประสิทธิภาพของโค้ด x86 SIMD และ ARM SVE โดยพื้นฐาน สามารถทำสิ่งต่าง ๆ เช่น การคลายลูป การ

เนื่องจาก SVE ไม่มีสิ่งที่เทียบเท่ากับ LMULทำให้ RVV ดำเนินการคำสั่งน้อยลง 6.3 เท่า โดยไม่ขึ้นกับขนาดของสถานะสถาปัตยกรรมเวกเตอร์ ในขณะที่ RVV ดำเนินการคำสั่งน้อยกว่า AVX2 หรือ AVX-512 สำหรับทุกขนาดสถานะเวกเตอร์ในตารางนี้ SVE ดำเนินการคำสั่งมากกว่า AVX2 หรือ AVX-512 1.3 เท่า เมื่อขนาดสถานะเวกเตอร์เท่ากับขนาดสถานะ AVX ความแตกต่างที่สำคัญที่สุดระหว่างสถาปัตยกรรม RVV และ SVE คือ:

- SVE ไม่มีตัวคูณความยาวเวกเตอร์ (`LMUL`) ที่เทียบเท่า ดังนั้นเรจิสเตอร์เวกเตอร์ทั้งหมดมีความยาวเท่ากัน (1/32 ของสถานะสถาปัตยกรรมเวกเตอร์) ไม่ว่าส่วนโค้ดจะต้องการเรจิสเตอร์จำนวนเท่าใด `LMUL` ลดจำนวนรอบการวนลูปของ DAXPY สำหรับ RVV ลง 8 เท่า

- การดำเนินการ SVE มีความกว้างของข้อมูลคงที่ ต่างจากการพิมพ์แบบไดนามิกของ RVV ดังนั้นสถาปัตยกรรม SVE จึงมีคำสั่งมากกว่ามาก

- SVE ลดค่าใช้จ่ายจากการแบ่งส่วนข้อมูล (strip-mining) โดยการตั้งค่ารีจิสเตอร์มาสก์ (predicate) ด้วยคำสั่ง `while` แทนการควบคุมความยาวเวกเตอร์ ด้วย `vsetvli` อย่างที่ RVV ทำ การพึ่งพามาสก์สำหรับการขุดแถบอาจส่งผลต่อประสิทธิภาพสำหรับลูปที่มีคำสั่งเงื่อนไข และยังทำให้การออกแบบโปรเซสเซอร์เวกเตอร์แบบชั่วคราว ซับซ้อนขึ้น (ที่ดำเนินการองค์ประกอบต่อเนื่องของเวกเตอร์ตามลำดับเวลาแทนที่จะพร้อมกัน)

- SVE ไม่รองรับการดำเนินการแบบเวกเตอร์-สเกลาร์โดยตรง ดังนั้นจึงต้องใช้คำสั่งพิเศษเพื่อทำซ้ำรีจิสเตอร์สเกลาร์ไปทั่วรีจิสเตอร์เวกเตอร์ (ดูตำแหน่งที่ 4 ใน รูปที่ 8.11)

- โหมดการกำหนดที่อยู่แบบปรับสเกลด้วยดัชนีของ ARMv8 SVE ลดจำนวนคำสั่งที่คำนวณที่อยู่ในลูปหลักของ DAXPY (ตำแหน่ง c, 10, และ 18)

■ ***รายละเอียดเพิ่มเติม: Illiac IV เป็นคอมพิวเตอร์เครื่องแรกที่แสดงให้เห็นถึงความยากลำบากในการคอมไพล์สำหรับ SIMD)***

ด้วยหน่วยประมวลผลจุดทศนิยม 64 บิตแบบขนาน 64 หน่วย (FPUs) Illiac IV ถูกวางแผนให้มีเกตตรรกะมากกว่า 1 ล้านเกต แม้กระทั่งก่อนที่Moore จะเผยแพร่กฎของเขา สถาปนิกของมันคาดการณ์ประสิทธิภาพเริ่มต้นที่ 1000 ล้านการดำเนินการจุดทศนิยมต่อวินาที (MFLOPS) David Kuck เป็นสถาปนิกซอฟต์แวรเพียงคนเดียวสำหรับ Illiac IV (ประมาณปี 1975) คำสะท้อนของเขา [Kuc 1991] คือ “สิ่งที่ฉันรู้สึกหงุดหงิดจริง ๆ คือการเขียนโปรแกรมสำหรับเครื่องนี้ยากมาก และสถาปัตยกรรมอาจไม่เหมาะกับบางแอปพลิเคชันที่เราพยายามรัน แนวคิดหลักคือผมไม่คิดว่าเรามีความเข้ากันได้ดีระหว่างแอปพลิเคชันและสถาปัตยกรรมใน Illiac IV” ประสิทธิภาพจริงของมันสูงสุดที่ 15 MFLOPS ค่าใช้จ่ายพุ่งสูงจากที่ประเมินไว้ 8 ล้านเหรียญในปี 1966 เป็น 31 ล้านเหรียญในปี 1972 แม้ว่าจะสร้างได้เพียง 64 จาก 256 FPUs ที่วางแผนไว้ โครงการเริ่มในปี 1965 แต่ใช้เวลาถึงปี 1976 กว่าจะรันแอปพลิเคชันจริงครั้งแรก ซึ่งเป็นปีที่ Cray-1 เปิดตัว บางทีอาจเป็นซูเปอร์คอมพิวเตอร์ที่มีชื่อที่ติดลบที่สุด มันติดอันดับ 10 รายการภัยพิบัติทางวิศวกรรม [Falk 1976]

8.13 ข้อสังเกตสรุป

หากโค้ดสามารถทำให้เป็นแบบเวกเตอร์ได้ สถาปัตยกรรมที่ดีที่สุดคือเวกเตอร์
—จิม สมิธ ปาฐกถาพิเศษ, การประชุมนานาชาติว่าด้วยสถาปัตยกรรมคอมพิวเตอร์ม, 1994

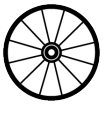
หากคุณเปลี่ยนมุมมอง คุณสามารถมอง SIMD เป็นสถาปัตยกรรมเวกเตอร์ที่มีข้อจำกัดด้วยรีจิสเตอร์เวกเตอร์สั้น ๆ ผลลัพธ์ของ DAXPY เผยให้เห็นการพัฒนาของสถาปัตยกรรม SIMD โดยมี AVX2 เป็นพื้นฐาน AVX-512 เพิ่มการใช้มาสก์ ทำให้สามารถทำให้เงื่อนไขและการแบ่งส่วนข้อมูล (strip-mining) เป็นเวกเตอร์ได้โดยไม่ต้องทำซ้ำเนื้อหาลูปสำหรับกรณีขอบ ขั้นตอนต่อไปของ SIMD คือการแยกความยาวเวกเตอร์ออกจากรหัสดำเนินการ เช่นใน SVE ตัวคูณความยาว (LMUL) ของ RVV เป็นการพัฒนาต่อไปในระดับวิวัฒนาการ ลดการรับส่งคำสั่งสำหรับการคำนวณเดียวกัน

เมื่อเปรียบเทียบผลลัพธ์ ในรูปที่ 8.12 กับรุ่นสเกลาร์ของ DAXPY ในรูปที่ 5.6 หน้าที่ 63 SIMD เพิ่มขนาด โค้ดเป็นสองเท่าทั้งในแง่คำสั่งและไบต์เมื่อเทียบกับโค้ดสเกลาร์ x86 แต่ลูปหลักดำเนินการคำสั่งจำนวนเท่าเดิม

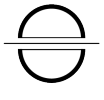

ประสิทธิภาพ

การลดจำนวนคำสั่งแบบไดนามิกที่ดำเนินการลดลงเกือบ 7 เท่าสำหรับ AVX-512 เมื่อเทียบกับโค้ดสเกลาร์ x86 ขนาดโค้ดเวกเตอร์ RV64GCV เพิ่มขึ้น 1.1 เท่าเมื่อเทียบกับสเกลาร์ RV64C แต่ด้วยสถานะสถาปัตยกรรมเวกเตอร์ 2048 ไบต์ จำนวนคำสั่งแบบไดนามิกของ RV64CV ลดลง 40 เท่าเมื่อเทียบกับ RV64C!

วิธีหนึ่งในการวัดสถาปัตยกรรมแบบขนานข้อมูลคือการหาค่า N_v ซึ่งในกรณีนี้คือค่า `n` ที่โค้ดแบบขนานข้อมูลทำงานเร็วกว่าโค้ดสเกลาร์ สมมติว่าไม่มีแคชมิสและทุกคำสั่งในโปรเซสเซอร์สมมติสามตัวของเราใช้หนึ่งรอบนาฬิกาเพื่อทำให้การคำนวณง่ายขึ้น ข้อยกเว้นเดียวคือคำสั่งเวกเตอร์ ซึ่งเราจะสมมติว่าสามารถประมวลผลสี่องค์ประกอบต่อรอบสัญญาณนาฬิกา แม้ว่าจำนวนคำสั่งแบบไดนามิกจะแตกต่างกันมาก แต่ในมุมมองของเรานั่นเป็นความแตกต่างที่สำคัญอันดับสองระหว่าง SIMD และเวกเตอร์ การขาดโมเดลการเขียนโปรแกรมที่ไม่ขึ้นกับความยาวเวกเตอร์ทำให้จำนวนคำสั่งเพิ่มขึ้นอย่างมาก ISA เช่น x86-64 ที่ยึดตามหลักการเพิ่มขั้น (incrementalist doctrine)—แทนที่จะเป็นปรัชญาแบบโมดูลาร์ของ RISC-V—ต้องทำซ้ำคำสั่ง SIMD เก่าทั้งหมดที่กำหนดไว้สำหรับรีจิสเตอร์ SIMD ที่แคบกว่าทุกครั้งที่เพิ่มความกว้าง SIMD เป็นสองเท่า แน่นอนว่ามีคำสั่ง x86 นับร้อยคำสั่งที่ถูกสร้างขึ้นผ่านหลายรุ่นของ ISA แบบ SIMD และยังมีอีกนับร้อยคำสั่งในอนาคต ภาระด้านความเข้าใจของโปรแกรมเมอร์ภาษาแอสเซมบลีจากแนวทางการพัฒนา ISA แบบใช้กำลังนี้จะหนักหน่วง ใครจะจำได้ว่า `vfmadd213pd` หมายถึงอะไร และควรใช้เมื่อใด


ความเรียบง่าย


การเขียนโปรแกรม


แยกสถาปัตย์ กับนำไปใช้

ในทางกลับกัน โค้ดเวกเตอร์ไม่ได้รับผลกระทบจากขนาดของเรจิสเตอร์เวกเตอร์ ไม่เพียงแต่โค้ดเวกเตอร์ไม่เปลี่ยนแปลงหากขนาดหน่วยความจำเวกเตอร์ขยาย คุณไม่จำเป็นต้องคอมไพล์ใหม่ด้วยซ้ำ เนื่องจากโปรเซสเซอร์ให้ค่าความยาวเวกเตอร์สูงสุด โค้ดในรูปที่ 8.4 ยังคงไม่เปลี่ยนแปลงหากโปรเซสเซอร์เพิ่มสถานะเวกเตอร์ จาก 512 ไบต์เป็น 2048 ไบต์

ต่างจาก SIMD ที่ ISA กำหนดฮาร์ดแวร์ที่ต้องการ—และการเปลี่ยน ISA หมายถึงการเปลี่ยนคอมไพเลอร์—ISA แบบเวกเตอร์ช่วยให้ผู้ออกแบบโปรเซสเซอร์เลือกทรัพยากรสำหรับการขนานข้อมูลสำหรับแอปพลิเคชันของตนได้โดยไม่กระทบต่อโปรแกรมเมอร์หรือคอมไพเลอร์ สามารถโต้แย้งได้ว่า SIMD ละเมิดหลักการออกแบบ ISA จากบทที่ 1 ที่แยกสถาปัตยกรรมออกจากการใช้งาน [Patterson and Waterman 2017]


สง่างาม

เราคิดว่าความแตกต่างสูงในด้านต้นทุน-พลังงาน-ประสิทธิภาพ ความซับซ้อน และความง่ายในการเขียนโปรแกรม ระหว่าง แนวทางเวกเตอร์แบบโมดูลาร์ของ RISC-V เทียบกับสถาปัตยกรรม SIMD แบบเพิ่มขั้นของ x86-64 และสถาปัตยกรรมเวกเตอร์ที่มีประสิทธิภาพต่ำกว่าของ SVE อาจเป็นหนึ่งในข้อโต้แย้งที่น่าเชื่อถือที่สุดสำหรับ RISC-V

■ ***รายละเอียดเพิ่มเติม: RV128V***

ตามที่อธิบายไว้ในส่วนท้ายของบทที่ 10 มีเวอร์ชันตำแหน่งที่อยู่ 128 บิต และเวอร์ชันตำแหน่งที่อยู่ 64 บิตของ RISC-V แม้ว่าจะไม่มีความจำเป็นเร่งด่วนสำหรับโปรเซสเซอร์เวกเตอร์ที่มีขนาดที่อยู่ใหญ่เช่นนั้น แต่ในช่วงอายุที่คาดหวังอันยาวนานของ RISC-V อาจมีความต้องการ การพิมพ์แบบไดนามิกหมายความว่าชุดคำสั่งในบทนี้รองรับ RV32V, RV64V และแม้แต่ RV128V

8.14 เรียนรู้เพิ่มเติม

Oral-history:David Kuck, 1991. URL `https://ethw.org/Oral-History:David_Kuck`.

H. Falk. What went wrong V: Reaching for a gigaflop: The fate of the famed Illiac IV was shaped by both research brilliance and real-world disasters. *IEEE Spectrum*, 13(10):65--70, 1976.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach, sixth edition*. Elsevier, 2019.

D. Patterson and A. Waterman. SIMD instructions considered harmful, 2017. URL `https://www.sigarch.org/simd-instructions-considered-harmful/`.

7. Vector Loads and Stores

7.4. Vector Unit-Stride Instructions

31 29	28	27 26	25	24 20	19 15	14 12	11 7	6 0	
000	0	00	vm	00000	rs1	000	**vd**	0000111	VLE8.V
000	0	00	vm	00000	rs1	101	**vd**	0000111	VLE16.V
000	0	00	vm	00000	rs1	110	**vd**	0000111	VLE32.V
000	0	00	vm	00000	rs1	111	**vd**	0000111	VLE64.V
000	0	00	vm	00000	rs1	000	**vs3**	0100111	VSE8.V
000	0	00	vm	00000	rs1	101	**vs3**	0100111	VSE16.V
000	0	00	vm	00000	rs1	110	**vs3**	0100111	VSE32.V
000	0	00	vm	00000	rs1	111	**vs3**	0100111	VSE64.V
000	0	00	1	01011	rs1	000	**vd**	0000111	VLM.V
000	0	00	1	01011	rs1	000	**vs3**	0100111	VSM.V

7.5. Vector Strided Instructions

31 29	28	27 26	25	24 20	19 15	14 12	11 7	6 0	
000	0	10	vm	rs2	rs1	000	**vd**	0000111	VLSE8.V
000	0	10	vm	rs2	rs1	101	**vd**	0000111	VLSE16.V
000	0	10	vm	rs2	rs1	110	**vd**	0000111	VLSE32.V
000	0	10	vm	rs2	rs1	111	**vd**	0000111	VLSE64.V
000	0	10	vm	rs2	rs1	000	**vs3**	0100111	VSSE8.V
000	0	10	vm	rs2	rs1	101	**vs3**	0100111	VSSE16.V
000	0	10	vm	rs2	rs1	110	**vs3**	0100111	VSSE32.V
000	0	10	vm	rs2	rs1	111	**vs3**	0100111	VSSE64.V

7.6. Vector Indexed Instructions

31 29	28	27 26	25	24 20	19 15	14 12	11 7	6 0	
000	0	01	vm	**vs2**	rs1	000	**vd**	0000111	VLUXEI8.V
000	0	01	vm	**vs2**	rs1	101	**vd**	0000111	VLUXEI16.V
000	0	01	vm	**vs2**	rs1	110	**vd**	0000111	VLUXEI32.V
000	0	01	vm	**vs2**	rs1	111	**vd**	0000111	VLUXEI64.V
000	0	01	vm	**vs2**	rs1	000	**vs3**	0100111	VSUXEI8.V
000	0	01	vm	**vs2**	rs1	101	**vs3**	0100111	VSUXEI16.V
000	0	01	vm	**vs2**	rs1	110	**vs3**	0100111	VSUXEI32.V
000	0	01	vm	**vs2**	rs1	111	**vs3**	0100111	VSUXEI64.V
000	0	11	vm	**vs2**	rs1	000	**vd**	0000111	VLOXEI8.V
000	0	11	vm	**vs2**	rs1	101	**vd**	0000111	VLOXEI16.V
000	0	11	vm	**vs2**	rs1	110	**vd**	0000111	VLOXEI32.V
000	0	11	vm	**vs2**	rs1	111	**vd**	0000111	VLOXEI64.V
000	0	11	vm	**vs2**	rs1	000	**vs3**	0100111	VSOXEI8.V
000	0	11	vm	**vs2**	rs1	101	**vs3**	0100111	VSOXEI16.V
000	0	11	vm	**vs2**	rs1	110	**vs3**	0100111	VSOXEI32.V
000	0	11	vm	**vs2**	rs1	111	**vs3**	0100111	VSOXEI64.V

รูปที่ 8.13: แผนผังรหัสดำเนินการ RV32V สำหรับการโหลดและการจัดเก็บหลัก มีเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ไม่แสดงหมายเลขบทในรูปเนื่องจากเป็นแบบจำลอง)

11.4. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
010000	0	**vs2**	**vs1**	000	**vd**	1010111	VADC.VVM
010000	0	**vs2**	rs1	101	**vd**	1010111	VADC.VXM
010000	0	**vs2**	simm5	011	**vd**	1010111	VADC.VIM
010001	0	**vs2**	**vs1**	000	**vd**	1010111	VMADC.VVM
010001	0	**vs2**	rs1	101	**vd**	1010111	VMADC.VXM
010001	0	**vs2**	simm5	011	**vd**	1010111	VMADC.VIM
010010	0	**vs2**	**vs1**	000	**vd**	1010111	VSBC.VVM
010010	0	**vs2**	rs1	101	**vd**	1010111	VSBC.VXM
010011	0	**vs2**	**vs1**	000	**vd**	1010111	VMSBC.VVM
010011	0	**vs2**	rs1	101	**vd**	1010111	VMSBC.VXM

11.5. Vector Bitwise Logical Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
001001	vm	**vs2**	**vs1**	000	**vd**	1010111	VAND.VV
001001	vm	**vs2**	rs1	101	**vd**	1010111	VAND.VX
001001	vm	**vs2**	simm5	011	**vd**	1010111	VAND.VI
001010	vm	**vs2**	**vs1**	000	**vd**	1010111	VOR.VV
001010	vm	**vs2**	rs1	101	**vd**	1010111	VOR.VX
001010	vm	**vs2**	simm5	011	**vd**	1010111	VOR.VI
001011	vm	**vs2**	**vs1**	000	**vd**	1010111	VXOR.VV
001011	vm	**vs2**	rs1	101	**vd**	1010111	VXOR.VX
001011	vm	**vs2**	simm5	011	**vd**	1010111	VXOR.VI

11.6. Vector Single-Width Bit Shift Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
100101	vm	**vs2**	**vs1**	000	**vd**	1010111	VSLL.VV
100101	vm	**vs2**	rs1	101	**vd**	1010111	VSLL.VX
100101	vm	**vs2**	simm5	011	**vd**	1010111	VSLL.VI
101000	vm	**vs2**	**vs1**	000	**vd**	1010111	VSRL.VV
101000	vm	**vs2**	rs1	101	**vd**	1010111	VSRL.VX
101000	vm	**vs2**	simm5	011	**vd**	1010111	VSRL.VI
101001	vm	**vs2**	**vs1**	000	**vd**	1010111	VSRA.VV
101001	vm	**vs2**	rs1	101	**vd**	1010111	VSRA.VX
101001	vm	**vs2**	simm5	011	**vd**	1010111	VSRA.VI

รูปที่ 8.14: แผนผังรหัสดำเนินการ RV32V สำหรับการดำเนินการเลขคณิตและตรรกะบางส่วน มีเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ไม่แสดงหมายเลขบทในรูป เนื่องจากเป็นแบบจำลอง)

11.1. Vector Single-Width Integer Add and Subtract

31–26	25	24–20	19–15	14–12	11–7	6–0	
000000	vm	**vs2**	**vs1**	000	**vd**	1010111	VADD.VV
000000	vm	**vs2**	rs1	101	**vd**	1010111	VADD.VX
000000	vm	**vs2**	simm5	011	**vd**	1010111	VADD.VI
000010	vm	**vs2**	**vs1**	000	**vd**	1010111	VSUB.VV
000010	vm	**vs2**	rs1	101	**vd**	1010111	VSUB.VX
000011	vm	**vs2**	rs1	101	**vd**	1010111	VRSUB.VX
000011	vm	**vs2**	simm5	011	**vd**	1010111	VRSUB.VI

11.8. Vector Integer Comparison Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
011000	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSEQ.VV
011000	vm	**vs2**	rs1	101	**vd**	1010111	VMSEQ.VX
011000	vm	**vs2**	simm5	011	**vd**	1010111	VMSEQ.VI
011001	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSNE.VV
011001	vm	**vs2**	rs1	101	**vd**	1010111	VMSNE.VX
011001	vm	**vs2**	simm5	011	**vd**	1010111	VMSNE.VI
011010	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSLTU.VV
011010	vm	**vs2**	rs1	101	**vd**	1010111	VMSLTU.VX
011011	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSLT.VV
011011	vm	**vs2**	rs1	101	**vd**	1010111	VMSLT.VX
011100	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSLEU.VV
011100	vm	**vs2**	rs1	101	**vd**	1010111	VMSLEU.VX
011100	vm	**vs2**	simm5	011	**vd**	1010111	VMSLEU.VI
011101	vm	**vs2**	**vs1**	000	**vd**	1010111	VMSLE.VV
011101	vm	**vs2**	rs1	101	**vd**	1010111	VMSLE.VX
011101	vm	**vs2**	simm5	011	**vd**	1010111	VMSLE.VI
011110	vm	**vs2**	rs1	101	**vd**	1010111	VMSGTU.VX
011110	vm	**vs2**	simm5	011	**vd**	1010111	VMSGTU.VI
011111	vm	**vs2**	rs1	101	**vd**	1010111	VMSGT.VX
011111	vm	**vs2**	simm5	011	**vd**	1010111	VMSGT.VI

11.13. Vector Single-Width Integer Multiply-Add Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
101101	vm	**vs2**	**vs1**	010	**vd / vs3**	1010111	VMACC.VV
101101	vm	**vs2**	rs1	101	**vd / vs3**	1010111	VMACC.VX
101111	vm	**vs2**	**vs1**	010	**vd / vs3**	1010111	VNMSAC.VV
101111	vm	**vs2**	rs1	101	**vd / vs3**	1010111	VNMSAC.VX
101001	vm	**vs2**	**vs1**	010	**vd / vs3**	1010111	VMADD.VV
101001	vm	**vs2**	rs1	101	**vd / vs3**	1010111	VMADD.VX
101011	vm	**vs2**	**vs1**	010	**vd / vs3**	1010111	VNMSUB.VV
101011	vm	**vs2**	rs1	101	**vd / vs3**	1010111	VNMSUB.VX

รูปที่ 8.15: แผนผังรหัสดำเนินการ RV32V สำหรับการดำเนินการเลขคณิตบางส่วน มีเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ไม่แสดงหมายเลขบทในรูป เนื่องจากเป็นแบบจำลอง)

13.11. Vector Floating-Point MIN/MAX Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
000100	vm	**vs2**	**vs1**	001	**vd**	1010111	VFMIN.VV
000100	vm	**vs2**	rs1	101	**vd**	1010111	VFMIN.VF
000110	vm	**vs2**	**vs1**	001	**vd**	1010111	VFMAX.VV
000110	vm	**vs2**	rs1	101	**vd**	1010111	VFMAX.VF

13.12. Vector Floating-Point Sign-Injection Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
001000	vm	**vs2**	**vs1**	001	**vd**	1010111	VFSGNJ.VV
001000	vm	**vs2**	rs1	101	**vd**	1010111	VFSGNJ.VF
001001	vm	**vs2**	**vs1**	001	**vd**	1010111	VFSGNJN.VV
001001	vm	**vs2**	rs1	101	**vd**	1010111	VFSGNJN.VF
001010	vm	**vs2**	**vs1**	001	**vd**	1010111	VFSGNJX.VV
001010	vm	**vs2**	rs1	101	**vd**	1010111	VFSGNJX.VF

13.13. Vector Floating-Point Compare Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
011000	vm	**vs2**	**vs1**	001	**vd**	1010111	VMFEQ.VV
011000	vm	**vs2**	rs1	101	**vd**	1010111	VMFEQ.VF
011100	vm	**vs2**	**vs1**	001	**vd**	1010111	VMFNE.VV
011100	vm	**vs2**	rs1	101	**vd**	1010111	VMFNE.VF
011011	vm	**vs2**	**vs1**	001	**vd**	1010111	VMFLT.VV
011011	vm	**vs2**	rs1	101	**vd**	1010111	VMFLT.VF
011001	vm	**vs2**	**vs1**	001	**vd**	1010111	VMFLE.VV
011001	vm	**vs2**	rs1	101	**vd**	1010111	VMFLE.VF
011101	vm	**vs2**	rs1	101	**vd**	1010111	VMFGT.VF
011111	vm	**vs2**	rs1	101	**vd**	1010111	VMFGE.VF

รูปที่ 8.16: รหัสดำเนินการ RV32V สำหรับการดำเนินการจุดทศนิยมบางส่วน มีเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ไม่แสดงหมายเลขบทในรูป เนื่องจากเป็นแบบจำลอง)

13. Vector Floating-Point Instructions

13.2. Vector Single-Width Floating-Point Add/Subtract Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
000000	vm	vs2	vs1	001	vd	1010111	VFADD.VV
000000	vm	vs2	rs1	101	vd	1010111	VFADD.VF
000010	vm	vs2	vs1	001	vd	1010111	VFSUB.VV
000010	vm	vs2	rs1	101	vd	1010111	VFSUB.VF
100111	vm	vs2	rs1	101	vd	1010111	VFRSUB.VF

13.4. Vector Single-Width Floating-Point Multiply/Divide Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
100100	vm	vs2	vs1	001	vd	1010111	VFMUL.VV
100100	vm	vs2	rs1	101	vd	1010111	VFMUL.VF
100000	vm	vs2	vs1	001	vd	1010111	VFDIV.VV
100000	vm	vs2	rs1	101	vd	1010111	VFDIV.VF
100001	vm	vs2	rs1	101	vd	1010111	VFRDIV.VF

13.6. Vector Single-Width Floating-Point Fused Multiply-Add Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
101100	vm	vs2	vs1	001	vd / vs3	1010111	VFMACC.VV
101100	vm	vs2	rs1	101	vd / vs3	1010111	VFMACC.VF
101101	vm	vs2	vs1	001	vd / vs3	1010111	VFNMACC.VV
101101	vm	vs2	rs1	101	vd / vs3	1010111	VFNMACC.VF
101110	vm	vs2	vs1	001	vd / vs3	1010111	VFMSAC.VV
101110	vm	vs2	rs1	101	vd / vs3	1010111	VFMSAC.VF
101111	vm	vs2	vs1	001	vd / vs3	1010111	VFNMSAC.VV
101111	vm	vs2	rs1	101	vd / vs3	1010111	VFNMSAC.VF
101000	vm	vs2	vs1	001	vd / vs3	1010111	VFMADD.VV
101000	vm	vs2	rs1	101	vd / vs3	1010111	VFMADD.VF
101001	vm	vs2	vs1	001	vd / vs3	1010111	VFNMADD.VV
101001	vm	vs2	rs1	101	vd / vs3	1010111	VFNMADD.VF
101010	vm	vs2	vs1	001	vd / vs3	1010111	VFMSUB.VV
101010	vm	vs2	rs1	101	vd / vs3	1010111	VFMSUB.VF
101011	vm	vs2	vs1	001	vd / vs3	1010111	VFNMSUB.VV
101011	vm	vs2	rs1	101	vd / vs3	1010111	VFNMSUB.VF

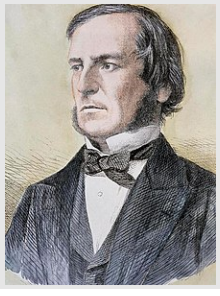
15.1. Vector Mask-Register Logical Instructions

31–26	25	24–20	19–15	14–12	11–7	6–0	
011001	vm	vs2	vs1	010	vd	1010111	VMAND.MM
011101	vm	vs2	vs1	010	vd	1010111	VMNAND.MM
011000	vm	vs2	vs1	010	vd	1010111	VMANDNOT.MM
011011	vm	vs2	vs1	010	vd	1010111	VMXOR.MM
011010	vm	vs2	vs1	010	vd	1010111	VMOR.MM
011110	vm	vs2	vs1	010	vd	1010111	VMNOR.MM
011100	vm	vs2	vs1	010	vd	1010111	VMORNOT.MM

รูปที่ 8.17: รหัสดำเนินการ RV32V สำหรับการดำเนินการจุดทศนิยมและมาสก์บางส่วน มีเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ไม่แสดงหมายเลขบทในรูป เนื่องจากเป็นแบบจำลอง)

9 RV32B: การจัดการบิต

จอร์จ บูล (1815–1864) เป็น นัก คณิตศาสตร์ ชาว อังกฤษ ที่ ที่ สนับสนุน ตรรกศาสตร์เชิงพีชคณิตและเป็นผู้ที่ตรรกศาสตร์บูลีนได้รับการตั้งชื่อตาม



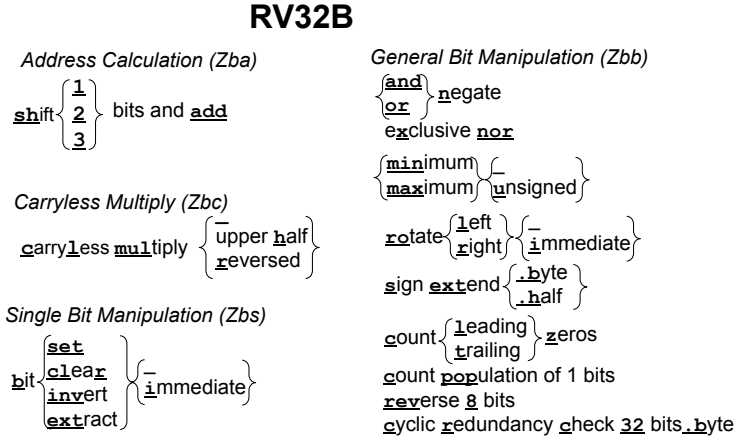
ไม่ว่าทฤษฎีคณิตศาสตร์จะดูเหมือนถูกต้องเพียงใด เราไม่ควรพอใจว่ามันสมบูรณ์แบบจนกว่าจะรู้สึกว่ามันสวยงามด้วย

—จอร์ท บูล, *ca.*1854

9.1 บทนำ

RV32B เพิ่มคำสั่งการจัดการบิตหลากหลายให้กับ RV32I [Marshall 2021] เช่นเดียวกับ RV32V คำสั่งเหล่านี้ครอบคลุมตั้งแต่คำสั่งที่คอมไพเลอร์มักสร้างขึ้น ไปจนถึงคำสั่งที่พบในโค้ดไลบรารีที่เขียนด้วยมือ การจัดการบิตยังเป็นส่วนขยายคำสั่งแรกที่ไม่เป็นแบบรวมศูนย์ (monolithic) โดยผู้ออกแบบโปรเซสเซอร์สามารถเลือกใช้ชุดย่อยของ RV32B ได้ ณ ปี 2022 ชุดย่อยอย่างเป็นทางการมี 4 ชุด ได้แก่:

- การคำนวณที่อยู่หน่วยความจำ เรียกว่า Zba
- การจัดการบิตทั่วไป, เรียกว่า Zbb
- การคูณแบบไม่ใช้แคร์รี่, เรียกว่า Zbc
- การจัดการบิตเดี่ยว, เรียกว่า Zbs



รูปที่ 9.1: แผนผังคำสั่ง RV32B

รูปที่ 9.1 เป็นการแสดงกราฟิกของชุดคำสั่งส่วนขยาย RV32B พร้อมระบุชุดย่อย และรูปที่ 9.2 แสดงรายการรหัสดำเนินการของคำสั่งเหล่านี้ ส่วนที่เหลือของบทนี้จะทบทวนชุดย่อยทั้งสี่นี้

9.2 Zba: การคำนวณที่อยู่หน่วยความจำ

คำสั่ง `sh1add`, `sh2add` และ `sh3add` จะเลื่อนบิตในรีจิสเตอร์ rs1 ไปทางซ้าย 1, 2 หรือ 3 บิต แล้วบวกค่าที่เลื่อนเข้ากับรีจิสเตอร์ rs2 การใช้งานที่ได้รับความนิยมมากที่สุดคือ การแปลงดัชนีของฮาล์ฟเวิร์ด (halfwords), เวิร์ด (words) และ ดับเบิลเวิร์ด (doublewords) เป็นที่อยู่หน่วยความจำแบบไบต์

9.3 Zbb: การจัดการบิตทั่วไป

ชุดย่อย RV32B นี้มีคำสั่งมากที่สุด คำสั่งเหล่านี้ครอบคลุมตั้งแต่คำสั่งที่พบทั่วไปในคอมพิวเตอร์อื่น ซึ่งคอมไพเลอร์จะสร้าง ไปจนถึงคำสั่งที่พบเฉพาะในไลบรารีที่เขียนด้วยมือและหาได้ยากในสถาปัตยกรรมอื่น

การดำเนินการตรรกะแบบกลับค่า คำสั่ง `andn`, `orn` และ `xnor` ดำเนินการตรรกะทั่วไป เช่น AND, OR และ XOR โดยตัวถูกดำเนินการที่สองจะถูกกลับค่า

ค่าสูงสุดและค่าต่ำสุด คำสั่ง `min`, `minu`, `max` และ `maxu` หาค่าต่ำสุดและสูงสุดระหว่างเรจิสเตอร์ต้นทางสองตัว โดยข้อมูลอาจเป็นจำนวนเต็มที่มีเครื่องหมายหรือไม่มีเครื่องหมาย

การหมุน คำสั่ง `rol` และ `ror` ทำการหมุนบิตไปทางซ้ายหรือขวาในรีจิสเตอร์ต้นทาง โดยจำนวนการหมุนระบุในรีจิสเตอร์อื่นหรือในฟิลด์ค่าคงที่ (immediate field)

การขยายเครื่องหมายและขยายศูนย์ คำสั่ง `sext.b` และ `sext.h` ขยายเครื่องหมายของไบต์ที่มีนัยสำคัญน้อยที่สุดหรือฮาล์ฟเวิร์ดในรีจิสเตอร์ให้เต็มความกว้าง คำสั่ง `zext.h` ขยายศูนย์ของไบต์ ที่มีนัยสำคัญน้อยที่สุดในรีจิสเตอร์ให้เต็มความกว้าง

การนับบิตศูนย์ คำสั่ง `clz` นับจำนวนศูนย์นำหน้า (leading zeros) ก่อนบิต 1 ตัวแรก โดยเริ่มจากบิตที่มีนัยสำคัญสูงสุด คำสั่ง `ctz` นับจำนวนศูนย์ต่อท้าย (trailing zeros) ก่อนบิต 1 ตัวแรก โดยเริ่มจากบิตที่มีนัยสำคัญน้อยที่สุด (least significant bit) ในทั้งสองกรณี ถ้าค่าที่ประเมินเป็น 0 ผลลัพธ์จะเท่ากับความกว้างของรีจิสเตอร์ และถ้าค่าเป็น -1 ผลลัพธ์จะเป็น 0 คำสั่งเหล่านี้ใช้สำหรับการสแกน bitmaps—เช่น ใน `malloc`—การหาค่าตัวหารร่วมมาก, คิวลำดับความสำคัญ, และไลบรารีซอฟต์แวร์ที่เลียนแบบการคำนวณจุดทศนิยม **การนับบิตหนึ่ง** คำสั่ง `popc` นับจำนวนบิต 1 ในรีจิสเตอร์ (population of 1s) ใช้สำหรับการคำนวณน้ำหนักแฮมมิง (Hamming weight) และอัลกอริทึมอื่น

การกลับลำดับและ CRC คำสั่ง `rev8` กลับลำดับของไบต์ในรีจิสเตอร์ เพื่อสลับระหว่างการแทนค่าแบบ big endian และ little endian คำสั่ง `CRC32.b` ดำเนินการลด

พหุนามของรีจิสเตอร์ที่ตีความเป็นสถานะ CRC32 และเลื่อน 8 บิต ใช้สำหรับการคำนวณ CRC ในโปรเซสเซอร์ RISC-V ที่ไม่มีคำสั่งคูณแบบไม่ใช้แคร์รี ซึ่งซึ่งจะกล่าวถึงต่อไป

9.4 Zbc: การคูณแบบไม่ใช้แคร์รี

การคูณแบบไม่ใช้แคร์รี (Carryless multiplication) เป็นการคูณในวงแหวนพหุนามเหนือ GF(2) (polynomial ring over GF(2)) ตัวอย่าง การใช้งาน รวมถึง การแฮช (hashing), การสร้างตัวเลขสุ่มเทียม (pseudo-random number generation), การคำนวณ Cyclic Redundancy Check (CRC), โหมด Galois/Counter (GCM), รหัส แก้ไข ข้อ ผิด พลาด (erasure codes), และรหัสเกรย์ (gray codes) การคูณแบบนี้คล้ายการคูณทั่วไปแต่ทิ้งแคร์รีออก (carry is discarded) หรือเรียกว่า *การคูณ XOR* เพราะการบวกแบบทิ้งแคร์รีเทียบเท่ากับการดำเนินการ XOR

คำสั่ง `clmul` สร้างครึ่งล่างของผลคูณแบบไม่ใช้แคร์รี และ `clmulh` สร้างครึ่งบน ถ้าความกว้างของรีจิสเตอร์เป็น XLEN คำสั่ง `clmulr` สร้างบิตตั้งแต่ 2 × XLEN − 2 ถึง XLEN − 1 ของผลคูณแบบไม่ใช้แคร์รี 2 × XLEN (หมายเหตุ: `clmulh` เท่ากับ `clmulr` ที่เลื่อนขวา 1 บิต)

9.5 Zbs: การจัดการบิตเดียว

คำสั่ง `bset`, `bclr`, `binv`, และ `bext` ตั้งค่าบิต, ล้างบิต, สลับค่าบิต, หรือดึงบิตเดียวในรีจิสเตอร์ตามดัชนีบิต (bit index) ที่ระบุในรีจิสเตอร์หรือในฟิลด์ค่าคงที่

9.6 ข้อสังเกตสรุป

> *หากคุณไม่สามารถลดปัญหาทางวิศวกรรมที่ยากให้เหลือเพียงกระดาษขนาด 8-1/2 x 11 นิ้ว ได้ คุณอาจจะไม่มีวันเข้าใจปัญหานั้น*
>
> —ราล์ฟ บราเซลตัน เพ็ค, วิศวกรโยธาที่มีชื่อเสียง



ประสิทธิภาพ

เพื่อรองรับการดำเนินการตรรกะทั้งแบบเรียบง่ายและซับซ้อน RV32B มีชุดคำสั่งหลากหลายให้สถาปนิก RISC-V เลือกใช้งาน

(ตารางที่ 2.1 ของ [Marshall 2021] เป็นที่มาของรูปนี้)

9.7 เรียนรู้เพิ่มเติม

B. Marshall, editor. *RISC-V Bit-Manipulation ISA-extensions Version 1.0.0*. June 2021. URL `https://github.com/riscv/riscv-bitmanip`.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0010000		rs2	rs1	010	rd	0110011	R sh1add
0010000		rs2	rs1	100	rd	0110011	R sh2add
0010000		rs2	rs1	110	rd	0110011	R sh3add
0000101		rs2	rs1	001	rd	0110011	R clmul
0000101		rs2	rs1	010	rd	0110011	R clmulr
0000101		rs2	rs1	011	rd	0110011	R clmulh
0100100		rs2	rs1	001	rd	0110011	R bclr
0010100		rs2	rs1	001	rd	0110011	R bset
0110100		rs2	rs1	001	rd	0110011	R binv
0100100		rs2	rs1	101	rd	0110011	R bext
01001	shamt		rs1	001	rd	0010011	I bclri
00101	shamt		rs1	001	rd	0010011	I bseti
01101	shamt		rs1	001	rd	0010011	I binvi
01001	shamt		rs1	101	rd	0010011	I bexti
0100000		rs2	rs1	111	rd	0110011	R andn
0100000		rs2	rs1	110	rd	0110011	R orn
0100000		rs2	rs1	100	rd	0110011	R xnor
0000101		rs2	rs1	100	rd	0110011	R min
0000101		rs2	rs1	101	rd	0110011	R minu
0000101		rs2	rs1	110	rd	0110011	R max
0000101		rs2	rs1	111	rd	0110011	R maxu
0110000		rs2	rs1	001	rd	0110011	R rol
0110000		rs2	rs1	101	rd	0110011	R ror
01100	shamt		rs1	101	rd	0010011	I rori
0110000		00100	rs1	001	rd	0010011	I sext.b
0110000		00101	rs1	001	rd	0010011	I sext.h
0000100		00000	rs1	100	rd	**011**0011	I zext.h
0110000		00000	rs1	001	rd	0010011	I clz
0110000		00001	rs1	001	rd	0010011	I ctz
0110000		00010	rs1	001	rd	0010011	I cpop
01101	0011000		rs1	101	rd	0010011	I rev8
not part of spec anymore (now Zbr)?							
0110000		10000	rs1	001	rd	0010011	I crc32.b

รูปที่ 9.2: แผนผังรหัสดำเนินการของ RV32B ประกอบด้วยโครงสร้างคำสั่ง (instruction layout), รหัสดำเนินการ (opcodes), ประเภทรูปแบบ (format type), และชื่อคำสั่ง (names)

10 RV64: คำสั่งที่อยู่หน่วยความจำ 64 บิต

มีข้อผิดพลาดเพียงข้อเดียวที่สามารถเกิดขึ้นในการออกแบบคอมพิวเตอร์ที่ยากต่อการแก้ไขได้—คือการไม่มีบิตที่อยู่เพียงพอสำหรับการกำหนดที่อยู่หน่วยความจำและการจัดการหน่วยความจำ

—ซี. กอร์ดอนเบลล์ ,1976

ซี. กอร์ดอน เบลล์ (1934–) เป็นหนึ่งในสถาปนิกหลักของ สถาปัตยกรรม มิ นิคอมพิวเตอร์ ที่ได้รับความนิยมมากที่สุดสองตัวในยุคนั้น: Digital Equipment Corporation PDP-11 (ที่อยู่ 16 บิต) ซึ่งเปิดตัวในปี 1970 และ ตัว ต่อ มา หลังจากนั้นเจ็ดปี Digital Equipment Corporation VAX-11 (Virtual Address eXtension) ที่อยู่ 32 บิต

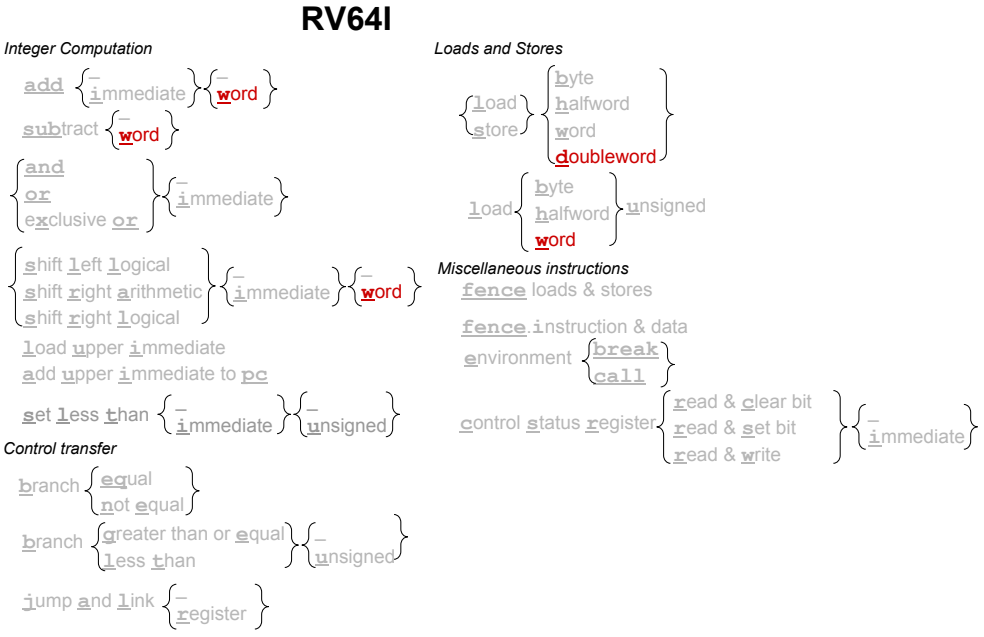


10.1 บทนำ

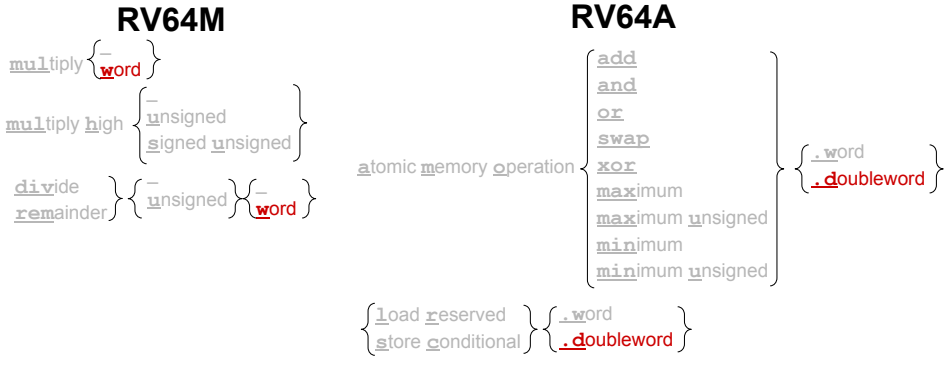
รูปที่ 10.1 ถึง 10.4 แสดงการนำเสนอแบบกราฟิกของคำสั่ง RV64G ซึ่งเป็นรุ่น 64 บิตของ คำสั่ง RV32G รูป เหล่า นี้ แสดง ให้ เห็น ถึง การ เพิ่ม คำ สั่ง เพียง เล็ก น้อย เพื่อ เปลี่ยนไปใช้สถาปัตยกรรมชุดคำสั่ง64 บิตใน RISC-V โดยทั่วไป ISA จะเพิ่มคำสั่งแบบ `word`, `doubleword` หรือ `long` จากคำสั่ง 32 บิต และขยายรีจิสเตอร์ทั้งหมด รวมถึงตัวนับโปรแกรม (Program Counter: PC) ให้เป็น 64 บิต ดังนั้น `sub` ใน RV64I จะลบจำนวน 64 บิตสองจำนวน แทนที่จะเป็นจำนวน 32 บิตสองจำนวนอย่างใน RV32I แม้ว่า RV64 เป็น ISA ที่ใกล้เคียงกันแต่แตกต่างจาก RV32 โดยเพิ่มคำสั่งใหม่เพียงไม่กี่คำสั่ง และคำสั่งพื้นฐานทำงานแตกต่างกันเล็กน้อย

ตัวอย่างเช่น การเรียงลำดับแบบแทรก (Insertion Sort) สำหรับ RV64I ในรูปที่ 10.10 มีความใกล้เคียงกับโค้ดสำหรับ RV32I ในรูปที่ 2.8 ในหน้า 34 ในบทที่ 2 โดยมีจำนวนคำสั่งและจำนวนไบต์เท่ากัน การเปลี่ยนแปลงเพียงอย่างเดียว คือคำสั่งโหลดและบันทึกเวิร์ด (word) กลายเป็นโหลดและบันทึกแบบดับเบิลเวิร์ด (doublewords) และการเพิ่มที่อยู่เปลี่ยนจาก 4 ไบต์สำหรับเวิร์ด เป็น 8 ไบต์สำหรับดับเบิลเวิร์ด รูปที่ 10.5 แสดงรายการรหัสดำเนินการ ของคำสั่ง RV64GC ในรูปที่ 10.1 ถึง 10.4

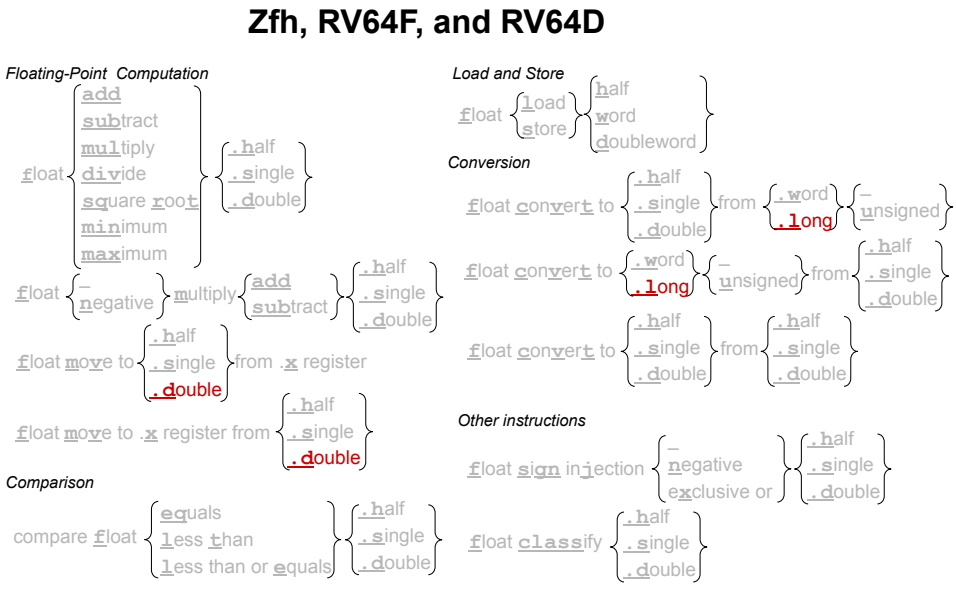
แม้ว่า RV64I จะใช้ที่อยู่หน่วยความจำ 64 บิต และขนาดข้อมูลเริ่มต้นเป็น 64 บิต แต่ข้อมูลแบบ 32 บิต (word) ยังคงเป็นประเภทข้อมูลที่ถูกต้องในโปรแกรม ดังนั้น RV64I ต้องรองรับ word เช่นเดียวกับที่ RV32I ต้องรองรับไบต์และฮาร์ฟเวิร์ด (half word) โดยเฉพาะอย่างยิ่ง เนื่องจาก รีจิสเตอร์มีความกว้าง 64 บิต RV64I จึงเพิ่มคำสั่งแบบเวิร์ดของคำสั่งบวกและลบ: `addw`, `addiw`, `subw` ซึ่งตัดผลลัพธ์ให้เหลือ 32 บิตและขยายเครื่องหมาย (sign-extend) ผลลัพธ์ไปยังรีจิสเตอร์ปลายทาง RV64I ยังมีคำสั่งเลื่อนบิต (shift) เพื่อได้ผลลัพธ์ของการเลื่อน 32 บิต แทน 64 บิต: `sllw`, `slliw`, `srlw`, `srliw`, `sraw`, `sraiw` สำหรับการรถ่ายโอนข้อมูล 64 บิต มีคำสั่งโหลดและบันทึกแบบดับเบิล



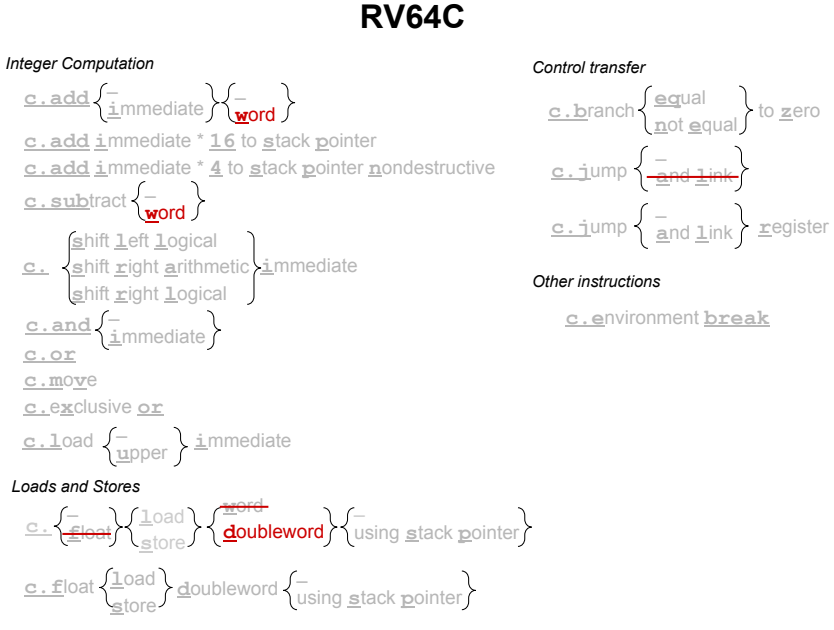
รูปที่ 10.1: แผนผังของชุดคำสั่ง RV64I ตัวอักษรที่ขีดเส้นใต้จะถูกต่อจากซ้ายไปขวาเพื่อสร้างคำสั่ง RV64I ส่วนที่สีจางประกอบด้วยคำสั่ง RV32I เดิม ซึ่งขยายให้ทำงานบนรีจิสเตอร์ 64 บิต ส่วนที่สีเข้ม (แดง) ประกอบด้วยคำสั่งใหม่สำหรับ RV64I



รูปที่ 10.2: แผนผังของคำสั่ง RV64M และ RV64A



รูปที่ 10.3: แผนผังของคำสั่ง Zfh, RV64F, และ RV64D



รูปที่ 10.4: แผนผังของคำสั่ง RV64C

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	110	rd	0000011	I lwu
imm[11:0]			rs1	011	rd	0000011	I ld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100011	S sd
000000		shamt	rs1	001	rd	0010011	I slli
000000		shamt	rs1	101	rd	0010011	I srli
010000		shamt	rs1	101	rd	0010011	I srai
imm[11:0]			rs1	000	rd	0011011	I addiw
0000000		shamt	rs1	001	rd	0011011	I slliw
0000000		shamt	rs1	101	rd	0011011	I srliw
0100000		shamt	rs1	101	rd	0011011	I sraiw
0000000		rs2	rs1	000	rd	0111011	R addw
0100000		rs2	rs1	000	rd	0111011	R subw
0000000		rs2	rs1	001	rd	0111011	R sllw
0000000		rs2	rs1	101	rd	0111011	R srlw
0100000		rs2	rs1	101	rd	0111011	R sraw

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	R mulw
0000001	rs2	rs1	100	rd	0111011	R divw
0000001	rs2	rs1	101	rd	0111011	R divuw
0000001	rs2	rs1	110	rd	0111011	R remw
0000001	rs2	rs1	111	rd	0111011	R remuw

RV64A Standard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	R lr.d
00011	aq	rl	rs2	rs1	011	rd	0101111	R sc.d
00001	aq	rl	rs2	rs1	011	rd	0101111	R amoswap.d
00000	aq	rl	rs2	rs1	011	rd	0101111	R amoadd.d
00100	aq	rl	rs2	rs1	011	rd	0101111	R amoxor.d
01100	aq	rl	rs2	rs1	011	rd	0101111	R amoand.d
01000	aq	rl	rs2	rs1	011	rd	0101111	R amoor.d
10000	aq	rl	rs2	rs1	011	rd	0101111	R amomin.d
10100	aq	rl	rs2	rs1	011	rd	0101111	R amomax.d
11000	aq	rl	rs2	rs1	011	rd	0101111	R amominu.d
11100	aq	rl	rs2	rs1	011	rd	0101111	R amomaxu.d

RV64F Standard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	R fcvt.l.s
1100000	00011	rs1	rm	rd	1010011	R fcvt.lu.s
1101000	00010	rs1	rm	rd	1010011	R fcvt.s.l
1101000	00011	rs1	rm	rd	1010011	R fcvt.s.lu

RV64D Standard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	R fcvt.l.d
1100001	00011	rs1	rm	rd	1010011	R fcvt.lu.d
1110001	00000	rs1	000	rd	1010011	R fmv.x.d
1101001	00010	rs1	rm	rd	1010011	R fcvt.d.l
1101001	00011	rs1	rm	rd	1010011	R fcvt.d.lu
1111001	00000	rs1	000	rd	1010011	R fmv.d.x

รูปที่ 10.5: แผนผังรหัสดำเนินการ RV64 สำหรับคำสั่งพื้นฐานและส่วนขยายเสริม แสดงเค้าโครงคำสั่ง รหัสดำเนินการ ประเภทรูปแบบ และชื่อ (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

เวิร์ด (doubleword) `ld`, `sd` สุดท้าย เช่นเดียวกับคำสั่งที่ไม่มีเครื่องหมายของโหลดไบต์ (byte) และฮาร์ฟเวิร์ด (halfword) ใน RV32I RV64I ต้องมีคำสั่งที่ไม่มีเครื่องหมายของการโหลดเวิร์ด `lwu`

ด้วยเหตุผลที่คล้ายกัน RV64M จำเป็นต้องเพิ่มคำสั่งแบบเวิร์ด (word) ของการคูณ การหาร และการหาเศษ: `mulw`, `divw`, `divuw`, `remw`, `remuw` เพื่อให้โปรแกรมเมอร์สามารถซิงโครไนซ์ได้ทั้งเวิร์ด (word) และดับเบิ้ลเวิร์ด (double word) RV64A เพิ่มคำสั่งแบบดับเบิ้ลเวิร์ด (double word) สำหรับคำสั่งทั้ง 11 คำสั่ง

Zfh, RV64F และ RV64D เพิ่ม จำนวนเต็ม ดับเบิ้ล เวิร์ด (doubleword) ให้ กับ คำ สั่ง แปลง (convert instructions) โดย เรียก ว่า *longs* เพื่อ ป้องกัน ความ สับสน กับ ข้อมูล จุดทศนิยม ความ แม่นยำ สอง เท่า: แปลง เป็น *long* แบบ มี เครื่องหมาย (`fcvt.lh`, `fcvt.ls`, `fcvt.ld`) แปลงเป็น long แบบไม่มีเครื่องหมาย (`fcvt.lu.h`, `fcvt.lu.s`, `fcvt.lu.d`) และ คำ สั่ง longs แบบ มี เครื่องหมาย (`fcvt.h.l`, `fcvt.s.l`, `fcvt.d.l`) และ จาก long แบบ ไม่มี เครื่องหมาย (`fcvt.h.lu`, `fcvt.s.lu`, `fcvt.d.lu`) เนื่องจากรีจิสเตอร์จำนวนเต็ม `x` มีความกว้าง 64 บิตแล้ว จึงสามารถเก็บข้อมูลจุดทศนิยมความแม่นยำสองเท่าได้ ดังนั้น RV64D จึงเพิ่มคำสั่งการย้ายจุดทศนิยมสองคำสั่ง: `fmv.x.d` และ `fmv.d.x`

ข้อยกเว้นเดียวสำหรับความสัมพันธ์แบบซูเปอร์เซ็ต (superset) ระหว่าง RV64 และ RV32 คือ คำสั่งแบบบีบอัด RV64C แทนที่คำสั่งบางตัวของ RV32C เนื่องจากคำสั่งอื่นช่วยลดขนาดโค้ดได้มากกว่าคำสั่งหน่วยความจำที่อยู่ 64 บิต RV64C ทิ้งคำสั่งกระโดดและเชื่อมโยงแบบบีบอัด (`c.jal`) และคำสั่งโหลดและจัดเก็บเวิร์ดสำหรับจำนวนเต็มและจุดทศนิยม (`c.lw`, `c.sw`, `c.lwsp`, `c.swsp`, `c.flw`, `c.fsw`, `c.flwsp.` `c.fswsp`) และแทนที่ด้วยคำสั่งการบวกและลบเวิร์ด ที่ได้รับความนิยมมากกว่า (`c.addw`, `c.addiw`, `c.subw`) และคำสั่งโหลดและบันทึกดับเบิ้ลเวิร์ด (`c.ld`, `c.sd`, `c.ldsp`, `c.sdsp`)


ขนาดโปรแกรม

รูปที่ 10.7 แสดงการนำเสนอแบบกราฟิกของส่วนขยาย RV64Zba, RV64Zbb และ RV64Zbs คล้าย กับ แผนภาพ ก่อน หน้า นี้ แต่ เพิ่ม คำ สั่ง เวิร์ด (word) ของ คำ สั่ง ที่ ใช้ ใน การ คำนวณ คำนวณ ตำแหน่ง ที่ อยู่ โดย ใช้ ตัวเลข ชดเชย แบบ ไม่มี เครื่องหมาย (unsigned offsets) ได้แก่ (`sh1add.uw`, `sh2add.uw`, `sh3add.uw`, `slli.uw`), การหมุน (`rolw`, `ror[i]w`), การนับ ได้แก่ (`clzw`, `ctzw`, `cpopw`) และการระดับบิตเดี่ยว ได้แก่ (`bclr[i]w`, `bset[i]w`, `binv[i]w`, `bextw`) ส่วน RV64Zbc ไม่เพิ่มคำสั่งใหม่ให้กับ RV32Zbc รูปที่ 10.6 รายการรหัสดำเนินการ RV64B

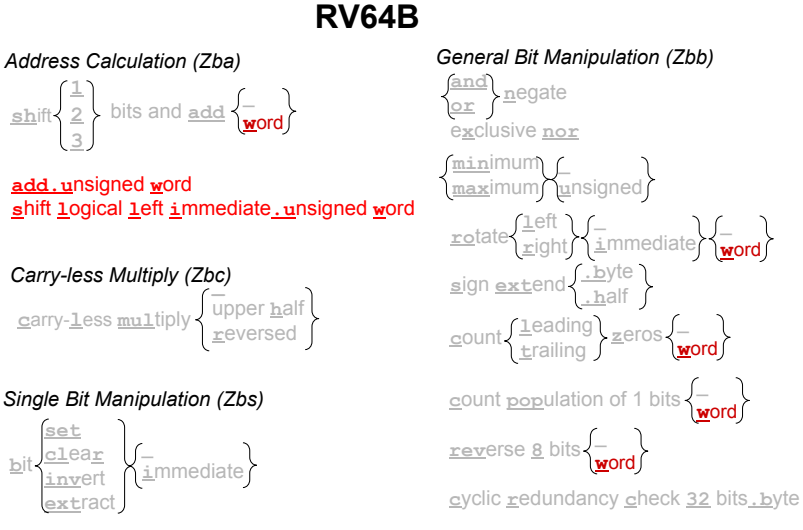
■ ***รายละเอียดเพิ่มเติม: ABI สำหรับ RV64 ประกอบด้วย lp64, lp64f, และ lp64d***

โดย lp64 หมายถึงประเภทข้อมูลในภาษา C ที่ long และ pointer มีขนาด 64 บิต ส่วน int ยังคงเป็น 32 บิต ตัวอักษรต่อท้าย f และ d ระบุวิธีการส่งผ่านอาร์กิวเมนต์จุดลอยตัว ซึ่งเหมือนกับใน RV32 (ดูบทที่ 3)

RV64B Standard Extension (in addition to RV32B)

31 25	24 20	19 15	14 12	11 7	6 0	
0010000	rs2	rs1	010	rd	0111011	R sh1add.uw
0010000	rs2	rs1	100	rd	0111011	R sh2add.uw
0010000	rs2	rs1	110	rd	0111011	R sh3add.uw
0110000	rs2	rs1	001	rd	0111011	R rolw
0110000	rs2	rs1	101	rd	0111011	R rorw
0110000	shamt	rs1	101	rd	0011011	I roriw
0110000	00000	rs1	001	rd	0011011	I clzw
0110000	00001	rs1	001	rd	0011011	I ctzw
0110000	00010	rs1	001	rd	0011011	I cpopw
0000100	rs2	rs1	000	rd	0111011	R add.uw
0000010	shamt	rs1	001	rd	0011011	I slli.uw

รูปที่ 10.6: แผนภาพ รหัสดำเนินการของส่วนขยายการจัดการบิตเสริมสำหรับ RV64 แสดงเค้าโครงคำสั่ง, รหัสดำเนินการ,ประเภทรูปแบบ และชื่อคำสั่ง (ตาราง 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)



รูปที่ 10.7: แผนภาพของคำสั่ง RV64B คำสั่ง `add.uw` และ `slli.uw` เหมือนกับคำสั่ง `add` และ `slli` ยกเว้นว่าส่วนครึ่งบนของตัวถูกดำเนินการ 64 บิต จะถูกล้างก่อนการบวกหรือการเลื่อนบิต

■ *รายละเอียดเพิ่มเติม: ไม่มีแผนภาพคำสั่งสำหรับ RV64V*

เนื่องจากมันเหมือนกับ RV32V ทุกประการ เพราะมีการเลือกความกว้างขององค์ประกอบแบบไดนามิก (dynamic element-width selection) การเปลี่ยนแปลงเพียงอย่างเดียวคือฟิลด์ `vill` ในรีจิสเตอร์ `vtype` ในรูปที่ 8.2 ย้ายไปที่บิต 63

10.2 การเปรียบเทียบกับ ISA 64 บิตอื่นโดยใช้การเรียงลำดับแบบแทรก

ตามที่กอร์ดอน เบลล์ กล่าวไว้ในตอนต้นของบทนี้ ข้อบกพร่องร้ายแรงของสถาปัตยกรรมคือการหมดบิตที่อยู่ เมื่อโปรแกรมเริ่มถึงขีดจำกัดของพื้นที่ที่อยู่ 32 บิต สถาปนิกจะเริ่มสร้างคำสั่งที่อยู่ 64 บิตของ ISA ของพวกเขา [Mashey 2009]

รุ่นแรกคือ MIPS ในปี 1991 ซึ่งขยายรีจิสเตอร์ทั้งหมดและตัวนับโปรแกรม (program counter) จาก 32 บิตเป็น 64 บิต และเพิ่มคำสั่ง 64 บิตของ MIPS-32 คำสั่งภาษาแอสเซมบลีของ MIPS-64 เริ่มต้นด้วยตัวอักษร “d” เช่น `daddu` หรือ `dsll` (ดูรูปที่ 10.12) โปรแกรมเมอร์สามารถผสมคำสั่ง MIPS-32 และ MIPS-64 ในโปรแกรมเดียวกันได้ MIPS-64 ยกเลิกช่องหน่วงการโหลด (load delay slot) จาก MIPS-32 (ซึ่งทำให้ไปป์ไลน์หยุดชะงักเมื่อมีการอ่านหลังเขียน read-after-write dependence)

การเขียนโปรแกรม

หนึ่งทศวรรษต่อมา ถึงเวลาสำหรับผู้สืบทอดของ x86-32 เมื่อสถาปนิกเพิ่มขนาดที่อยู่ พวกเขาใช้โอกาสนี้ปรับปรุง x86-64 เพิ่มเติมดังนี้:

- เพิ่มจำนวนรีจิสเตอร์จำนวนเต็มจาก 8 เป็น 16 (`r8–r15`);
- เพิ่มจำนวนรีจิสเตอร์ SIMD จาก 8 เป็น 16 (`xmm8–xmm15`);
- เพิ่มการกำหนดที่อยู่ข้อมูลสัมพันธ์กับ PC (PC-relative data addressing) เพื่อสนับสนุนโค้ดที่ไม่ขึ้นกับตำแหน่ง (position-independent code) ได้ดียิ่งขึ้น

การปรับปรุงเหล่านี้ช่วยขจัดข้อบกพร่องบางประการของ x86-32



ประสิทธิภาพ

คุณสามารถเห็นประโยชน์ได้โดยเปรียบเทียบ x86-32 ของการเรียงลำดับแบบแทรกในรูปที่ 2.11 หน้าที่ 37 บทที่ 2 กับรุ่น x86-64 ในรูปที่ 10.13 ISA รุ่นใหม่ เก็บตัวแปรทั้งหมดไว้ในรีจิสเตอร์ แทนการเก็บบางตัวในหน่วยความจำ ซึ่งลดจำนวนคำสั่งจาก 20 เป็น 15 คำสั่ง ขนาดโค้ดจริง ๆ แล้วใหญ่ขึ้นหนึ่งไบต์ใน ISA รุ่นใหม่ แม้ว่าจะมีจำนวนคำสั่งน้อยลง: 46 เทียบกับ 45 ไบต์ สาเหตุคือเพื่อให้รองรับรหัสคำสั่ง (opcodes) ใหม่สำหรับรีจิสเตอร์ที่เพิ่มขึ้น x86-64 ได้เพิ่มไบต์คำนำ (prefix byte) เพื่อระบุคำสั่งใหม่ ความยาวเฉลี่ยของคำสั่งใน x86-64 จึงเพิ่มขึ้นเมื่อเทียบกับ x86-32

การเขียนโปรแกรม

ARM เผชิญปัญหาที่อยู่เช่นเดียวกันในอีกหนึ่งทศวรรษต่อมา แทนที่จะพัฒนา ISA เดิมให้มีที่อยู่ 64 บิตเหมือน x86-64 พวกเขาใช้โอกาสนี้สร้าง ISA ใหม่ทั้งหมด ด้วยการเริ่มต้นใหม่ ARM ได้เปลี่ยนแปลงลักษณะที่ยุ่งยากของ ARM-32 เพื่อให้ได้ ISA ที่ทันสมัย:

- เพิ่มจำนวนรีจิสเตอร์จำนวนเต็มจาก 15 เป็น 31

ISA	ARM-64	MIPS-64	x86-64	RV64I	RV64I+RV64C
Instructions	16	24	15	19	19
Bytes	64	96	46	76	52

รูปที่ 10.8: จำนวนคำสั่งและขนาดโค้ดสำหรับการเรียงลำดับแบบแทรกในสี่ ISA ARM Thumb-2 และ microMIPS เป็น ISA ที่ใช้ที่อยู่ 32 บิต จึงไม่สามารถใช้ได้กับ ARM-64 และ MIPS-64

- ลบตัวนับโปรแกรม (PC) ออกจากชุดรีจิสเตอร์
- จัดให้มีรีจิสเตอร์ที่ตั้งค่าเป็นศูนย์อย่างถาวรสำหรับคำสั่งส่วนใหญ่ (`r31`);
- ต่างจาก ARM-32 โหมดการกำหนดที่อยู่ข้อมูลใน ARM-64 ทำงานได้กับทุกขนาดและประเภทของข้อมูล
- ARM-64 ยกเลิกคำสั่งโหลดและบันทึกหลายรายการของ ARM-32
- ARM-64 ไม่รวมตัวเลือกการทำงานแบบมีเงื่อนไขของคำสั่ง ARM-32

อย่างไรก็ตาม ARM-64 ยังคงมีจุดอ่อนบางประการจาก ARM-32 เช่น รหัสเงื่อนไขสำหรับการกระโดด (branch), ฟิลด์รีจิสเตอร์ต้นทางและปลายทางที่เปลี่ยนตำแหน่งในรูปแบบคำสั่ง, คำสั่งย้ายข้อมูลแบบมีเงื่อนไข (conditional move), โหมดการกำหนดที่อยู่ที่ซับซ้อน, ตัวนับประสิทธิภาพที่ไม่สม่ำเสมอ, และคำสั่งที่มีความยาวเพียง 32 บิต ARM-64 ไม่สามารถเปลี่ยนไปใช้ ISA แบบ Thumb-2 ได้ เนื่องจาก Thumb-2 ทำงานได้เฉพาะกับที่อยู่ 32 บิต

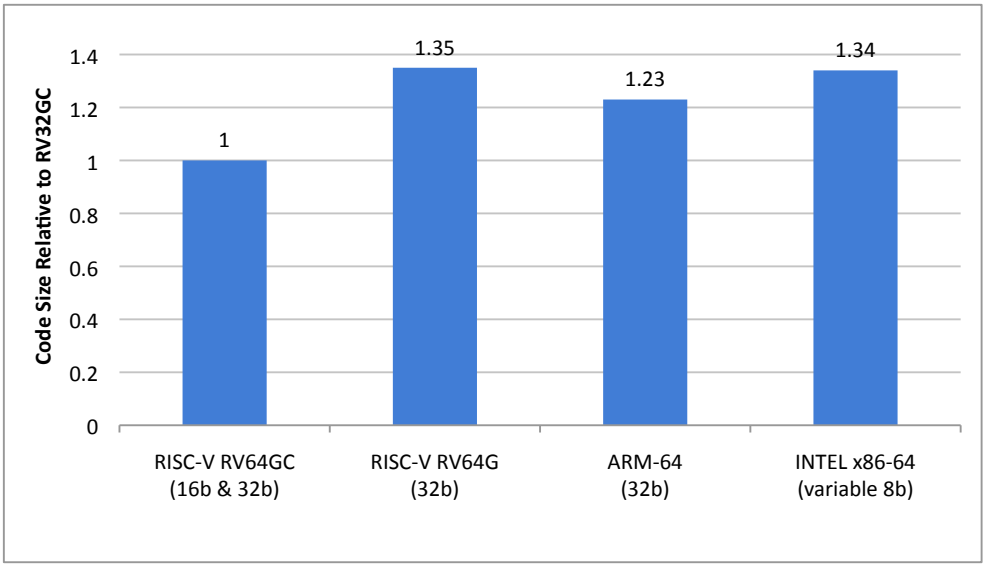
ต่างจาก RISC-V ที่ ARM เลือกใช้แนวทางการออกแบบ ISA แบบสูงสุด (maximalist approach) แม้ว่า ARM-64 จะเป็น ISA ที่ดีกว่า ARM-32 อย่างแน่นอน แต่ก็มีขนาดใหญ่กว่า ตัวอย่างเช่น มีคำสั่งมากกว่า 1,000 คำสั่ง และคู่มือ ARM-64 มีความยาวถึง 3,185 หน้า [ARM 2015] ยิ่งไปกว่านั้น มันยังคงขยายตัวอย่างต่อเนื่อง โดยมีการเพิ่มส่วนขยายของ ARM-64 มากมายนับตั้งแต่การประกาศในปี 2011

Intel ไม่ได้เป็นผู้สร้าง ISA แบบ x86-64 เมื่อเปลี่ยนไปใช้ที่อยู่ 64 บิต Intel ได้สร้าง ISA ใหม่ชื่อ Itanium ซึ่งไม่เข้ากันได้กับ x86-32 คู่แข่งของโปรเซสเซอร์ x86-32 ถูกกีดกันจาก Itanium ดังนั้น AMD จึงสร้างรุ่น 64 บิตของ x86-32 ชื่อ AMD64 Itanium ล้มเหลวในที่สุด ทำให้ Intel ต้องยอมรับ ISA แบบ AMD64 เป็นผู้สืบทอดที่อยู่ 64 บิตของ x86-32 ซึ่งเราเรียกว่า x86-64 [Kerner and Padgett 2007]

โค้ด ARM-64 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 10.11 ดูคล้ายโค้ด RV64I หรือ x86-64 มากกว่าโค้ด ARM-32 ตัวอย่างเช่น ด้วยรีจิสเตอร์ 31 ตัว ไม่จำเป็นต้องบันทึกและเรียกคืนรีจิสเตอร์จากสแต็ก และเนื่องจาก PC ไม่ใช่รีจิสเตอร์อีกต่อไป ARM-64 จึงใช้คำสั่งคืนค่าแยกต่างหาก

รูปที่ 10.8 เป็นตารางที่สรุปจำนวนคำสั่งและจำนวนไบต์ในโค้ดการเรียงลำดับการแทรก สำหรับ ISA ทั้งสี่ รูปที่ 10.10 ถึง 10.13 แสดงโค้ดที่คอมไพล์สำหรับ RV64I, ARM-64, MIPS-64 และ x86-64 ข้อความในวงเล็บในความคิดเห็นของโปรแกรมทั้งสี่นี้ระบุความแตกต่างระหว่างคำสั่ง RV32I ในบทที่ 2 และคำสั่ง RV64I เหล่านี้

MIPS-64 ต้องการจำนวนคำสั่งมากที่สุด ส่วนใหญ่เนื่องจากคำสั่ง `nop` ของช่องหน่วงการกระโดด (delayed branch slots) ที่ไม่ได้ถูกเติม RV64I ใช้คำสั่งน้อยกว่า เนื่องจากมีคำสั่งเปรียบเทียบและกระโดด (compare-and-branch) และไม่มีช่องหน่วงการกระ



รูปที่ 10.9: ขนาดโปรแกรมสัมพัทธ์สำหรับ RV64G, ARM-64 และ x86-64 เทียบกับ RV64GC การเปรียบเทียบนี้วัดขนาดโปรแกรมที่ใหญ่กว่ามาก เมื่อเทียบกับในรูปที่ 10.8 กราฟนี้เป็นรุ่นที่อยู่ 64 บิตที่เทียบเท่ากับกราฟของ ISA 32 บิต ในรูปที่ 1.5 ในหน้าที่ 11 บทที่ 1 ขนาดโค้ดของ RV32C เกือบจะเท่ากับ RV64C โดยเล็กกว่าประมาณ 1% ไม่มีตัวเลือก Thumb-2 สำหรับ ARM-64 ดังนั้นขนาดโค้ดของ ISA 64 บิตอื่นๆ จึงใหญ่กว่าโค้ด RV64GC อย่างมีนัยสำคัญ โปรแกรมที่วัดคือชุดการทดสอบ SPEC CPU2006 โดยใช้คอมไพเลอร์ GCC [Waterman 2016]

โดด ในขณะที่ ARM-64 และ x86-64 ต้องการคำสั่งเปรียบเทียบสองคำสั่งที่ไม่จำเป็นสำหรับ RV64I แต่โหมดการกำหนดที่อยู่แบบปรับสเกล (scaling addressing modes) ช่วยหลีกเลี่ยงคำสั่งคำนวณที่อยู่ที่จำเป็นใน RV64I ทำให้มีคำสั่งน้อยที่สุด อย่างไรก็ตาม RV64I+RV64C มีขนาดโค้ดที่เล็กกว่ามาก ดังที่ส่วนถัดไปจะอธิบาย

■ ***รายละเอียดเพิ่มเติม: ARM-64, MIPS-64 และ x86-64 ไม่ใช่ชื่ออย่างเป็นทางการ***

ชื่อ อย่าง เป็น ทางการ ของ ARMv9 คือ ARM-64 [ARM 2021] MIPS-IV คือ MIPS-64 และ AMD64 คือ x86-64 (ดูแถบด้านข้างหน้าก่อนหน้าสำหรับประวัติของ x86-64)

10.3 ขนาดโปรแกรม



รูป ที่ 10.9 เปรียบ เทียบ ขนาด โค้ด สัมพัทธ์ เฉลี่ย สำหรับ RV64, ARM-64 และ x86-64 เปรียบเทียบรูปนี้กับรูปที่ 1.5 ในหน้า 11 ในบทที่ 1. ประการแรก ขนาดโค้ด RV32GC มีขนาดเกือบเท่ากับ RV64GC โดยเล็กกว่าเพียง 1% ความใกล้เคียงนี้ยังใช้ได้กับ RV32I และ RV64I ในขณะที่โค้ด ARM-64 เล็กกว่าโค้ด ARM-32 ถึง 8% แต่ไม่มีรุ่นที่อยู่ 64 บิตของ Thumb-2 ดังนั้นคำสั่งทั้งหมดยังคงยาว 32 บิต ดังนั้นโค้ด ARM-64 *ใหญ่กว่า*โค้ด ARM Thumb-2 ถึง 25% โค้ด x86-64 ใหญ่ กว่า โค้ด x86-32 ถึง 7% เนื่องจาก การ

เพิ่มรหัสคำนำ (prefix opcodes) ในคำสั่ง x86-64 เพื่อรองรับการดำเนินการใหม่และชุดเรจิสเตอร์ที่ขยายออก RV64GC ชนะ โดยโค้ด ARM-64 ใหญ่กว่า RV64GC ถึง 23% และโค้ท x86-64 ใหญ่กว่า RV64GC ถึง 34% ความแตกต่างนี้มากพอที่จะช่วยปรับปรุงประสิทธิภาพโดยลดอัตราการพลาดของแคช คำสั่งที่ต่ำลงหรือลดต้นทุนโดยใช้แคชคำสั่งขนาดเล็กที่ยังคงให้อัตราการพลาดที่น่าพอใจ



ค่าใช้จ่าย

10.4 ข้อสรุปท้ายบท

ปัญหาอย่างหนึ่งของการเป็นผู้บุกเบิกคือ คุณมักจะทำผิดพลาดเสมอ และฉันไม่เคยอยากเป็นผู้บุกเบิกเลย สิ่งดีที่สุดคือการเป็นที่สอง เพื่อที่จะได้มองเห็นข้อผิดพลาดที่ผู้บุกเบิกทำ

—Seymour Cray สถาปนิกของซูเปอร์คอมพิวเตอร์เครื่องแรก, 1976

การ หมด จำนวน บิต ของ ที่ อยู่ (address bits) เป็น จุด อ่อน สำคัญ ของ สถาปัตยกรรมคอมพิวเตอร์ สถาปัตยกรรมหลายตัวต้องล่มสลายจากบาดแผลนี้ ARM-32 และ Thumb-2 ยังคงเป็นสถาปัตยกรรม 32 บิต ดังนั้นจึงไม่ช่วยสำหรับโปรแกรมขนาดใหญ่ ISA บางตัว เช่น MIPS-64 และ x86-64 รอดจากการเปลี่ยนผ่านนี้ แต่ x86-64 ไม่ใช่ตัวอย่างที่ดีของการออกแบบ ISA และอนาคตของ MIPS-64 ก็ยังไม่แน่นอนในขณะนี้ ARM-64 เป็น ISA ขนาดใหญ่ตัวใหม่ที่ใช้กันอย่างแพร่หลายในตลาดสมาร์ทโฟน และเริ่มประสบความสำเร็จในระบบคลาวด์ เมื่อมองไปข้างหน้า ชุดคำสั่ง 64 บิตที่ได้รับความนิยมมากที่สุดน่าจะเป็น ARM-64, RV64, และ x86-64

MIPS มี เจ้าของ ราย ที่ ห้า Tallwood Venture Capital ซึ่งซื้อ MIPS ในปี 2017 ด้วยมูลค่า 65 ล้านดอลลาร์สหรัฐ และกลายเป็นเจ้าของรายที่สาม ได้โอน MIPS ไปให้สตาร์ทอัพ Wave Computing ในปี 2018 เมื่อ Wave Computing ยื่นขอพิทักษ์ทรัพย์ ตาม Chapter 11 ในปี 2020 Tallwood Venture Capital ซื้อ MIPS คืนในราคา 61 ล้านดอลลาร์สหรัฐ



การเขียนโปรแกรม



ที่ว่างสำหรับส่วนขยาย



ขนาดโปรแรม



สง่างาม

RISC-V ได้รับประโยชน์จากการออกแบบสถาปัตยกรรม 32 บิตและ 64 บิตพร้อมกัน ในขณะที่ ISA รุ่นเก่าต้องออกแบบตามลำดับ ไม่น่าแปลกใจที่การเปลี่ยนผ่านระหว่าง 32 บิตและ 64 บิตนั้นง่ายที่สุดสำหรับโปรแกรมเมอร์และผู้เขียนคอมไพเลอร์ของ RISC-V โดย RV64I มีคำสั่งเกือบทั้งหมดของ RV32I นั่นคือเหตุผลที่เราสามารถระบุทั้ง RV32GCV และ RV64GCV ได้ในเพียงสองหน้าในบัตรอ้างอิง (Reference Card)

ทำให้สถาปัตยกรรม 64 บิตไม่ต้องถูกบีบอัดลงในพื้นที่รหัสคำสั่ง (opcode space) 32 บิตที่คับแคบ RV64I มีพื้นที่ว่างเพียงพอสำหรับส่วนขยายคำสั่งเสริม ซึ่งทำให้ RV64 เป็นผู้นำด้านขนาดโค้ด เรามองว่าสถาปัตยกรรม 64 บิตเป็นหลักฐานเพิ่มเติมของการออกแบบที่ยอดเยี่ยมของ RISC-V ซึ่งยอมรับว่าง่ายกว่าหากเริ่มต้นช้ากว่า 20 ปี เพื่อยืมแนวคิดที่ดีของผู้บุกเบิกและเรียนรู้จากข้อผิดพลาดของพวกเขา

■ ***รายละเอียดเพิ่มเติม: RV64E ส่วนขยายมาตรฐานสำหรับงานฝังตัว***

เช่นเดียวกับที่ RV32E ลดต้นทุนของคอร์ที่อยู่ 32 บิตระดับล่าง RV64E เป็นชุดคำสั่งพื้นฐานที่มีรีจิสเตอร์น้อยลง 16 ตัวสำหรับคอร์ที่อยู่ 64 บิต ระดับล่าง

■ ***รายละเอียดเพิ่มเติม: RV128***

RV128 เริ่มต้นจากมุกตลกภายในของสถาปนิก RISC-V เพื่อแสดงว่า ISA ที่อยู่ 128 บิตเป็นไปได้ อย่างไรก็ตาม คอมพิวเตอร์ขนาดใหญ่ระดับคลังสินค้า (warehouse scale computers) อาจมีหน่วยความจำเซมิคอนดักเตอร์ (DRAM และ Flash memory) 2^{64} ไบต์ในไม่ช้า ซึ่งโปรแกรมเมอร์อาจต้องการเข้าถึงที่อยู่หน่วยความจำ นอกจากนี้ยังมีข้อเสนอให้ใช้ที่อยู่ 128 บิตเพื่อปรับปรุงความปลอดภัย [Woodruff et al. 2014] คู่มือ RISC-V ระบุ ISA 128 บิตเต็มรูปแบบชื่อ RV128G [Waterman and Asanović 2017] คำสั่งเพิ่มเติมนั้นโดยพื้นฐานเหมือนกับที่ต้องใช้ในการเปลี่ยนจาก RV32 เป็น RV64 ซึ่ง แสดงในรูปที่ 10.1 ถึง 10.4 แสดงให้เห็น รีจิสเตอร์ทั้งหมดขยายเป็น 128 บิต และคำสั่ง RV128 ใหม่ระบุรุ่น 128 บิตของบางคำสั่ง (ใช้ Q ในชื่อสำหรับ `quadword`) หรือรุ่น 64 บิตของคำสั่งอื่น (ใช้ D ในชื่อสำหรับ `doubleword`)

10.5 เรียนรู้เพิ่มเติม

I. ARM. ARMv8-A architecture reference manual. 2015.

I. ARM. Arm architecture reference manual supplement armv9, for armv9-a architecture profile. 2021.

M. Kerner and N. Padgett. A history of modern 64-bit computing. Technical report, CS Department, University of Washington, Feb 2007. URL `http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf`.

J. Mashey. The long road to 64 bits. *Communications of the ACM*, 52(1): 45--53, 2009.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457--468. IEEE, 2014.

ในหน้าที่ 37 บทที่ 2 เนื่องจากเป็นชุดคำสั่งใหม่ รีจิสเตอร์เริ่มต้นด้วย x แทน a โหมดการกำหนดที่อยู่ข้อมูล (data addressing modes) สามารถเลื่อนรีจิสเตอร์ได้ 3 บิตเพื่อปรับสเกลดัชนีให้เป็นที่อยู่แบบไบต์ (byte address) ด้วยรีจิสเตอร์ 31 ตัว ไม่จำเป็นต้อง

```
# RV64I (19 คำสั่ง, 76 bytes, or 52 bytes กับ RV64C)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
   0: 00850693  addi  a3,a0,8    # (8 vs 4) a3 เป็นตัวชี้ไปที่ a[i]
   4: 00100713  li    a4,1       # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # ถ้า i < n กระโดดไปที่ Continue Outer loop
Exit Outer Loop:
   c: 00008067  ret              # ออกจากฟังก์ชัน
Continue Outer Loop:
  10: 0006b803  ld    a6,0(a3)   # (ld vs lw) x = a[i]
  14: 00068613  mv    a2,a3      # a2 เป็นตัวชี้ไปที่ a[j]
  18: 00070793  mv    a5,a4      # j = i
Inner Loop:
  1c: ff863883  ld    a7,-8(a2)  # (ld vs lw, 8 vs 4) a7 = a[j-1]
  20: 01185a63  ble   a7,a6,34   # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
  24: 01163023  sd    a7,0(a2)   # (sd vs sw) a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ff860613  addi  a2,a2,-8   # (8 vs 4) ลด a2 ชี้ไปที่ a[j]
  30: fe0796e3  bnez  a5,1c      # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
  34: 00379793  slli  a5,a5,0x3  # (8 vs 4) คูณ a5 by 8
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address of a[j]
  3c: 0107b023  sd    a6,0(a5)   # (sd vs sw) a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00868693  addi  a3,a3,8    # เพิ่ม a3 ชี้ไปที่ a[i]
  48: fc1ff06f  j     8          # กระโดดไปที่ Outer Loop
```

รูปที่ 10.10: โค้ด RV64I สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 โปรแกรมภาษาแอสเซมบลี RV64I คล้ายกับภาษาแอสเซมบลี RV32I ในรูปที่ 2.8 หน้าที่ 34 ในบทที่ 2 โดยเราระบุความแตกต่างไว้ในวงเล็บในความคิดเห็น (comments) ขนาดของข้อมูลเปลี่ยนจาก 4 ไบต์เป็น 8 ไบต์ ดังนั้นคำสั่งสามคำสั่งเปลี่ยนค่าคงที่จาก 4 เป็น 8 ความกว้างที่เพิ่มขึ้นนี้ยังเปลี่ยนคำสั่งโหลดเวิร์ด 2 คำสั่ง (`lw`) เป็นโหลดดับเบิลเวิร์ด (`ld`) และคำสั่งบันทึกเวิร์ด 2 คำสั่ง (`sw`) เป็นบันทึกดับเบิลเวิร์ด(`sd`)

```
# ARM-64 (16 คำสั่ง, 64 bytes)
# x0 ชี้ไปที่ a[0], x1 คือ n, x2 คือ j, x3 คือ i, x4 คือ x
  0: d2800023  mov  x3, #0x1             # i = 1
Outer Loop:
  4: eb01007f  cmp  x3, x1               # เปรียบเทียบ i vs n
  8: 54000043  b.cc 10                   # ถ้า i < n กระโดดไปที่ Continue Outer loop
Exit Outer Loop:
  c: d65f03c0  ret                       # ออกจากฟังก์ชัน
Continue Outer Loop:
 10: f8637804  ldr  x4, [x0, x3, lsl #3] # (x4 ca r4) vs x = a[i]
 14: aa0303e2  mov  x2, x3               # (x2 vs r2) j = i
Inner Loop:
 18: 8b020c05  add  x5, x0, x2, lsl #3   # x5 ตัวชี้ไปที่ a[j]
 1c: f85f80a5  ldur x5, [x5, #-8]        # x5 = a[j]
 20: eb0400bf  cmp  x5, x4               # เปรียบเทียบ a[j-1] vs. x
 24: 5400008d  b.le 34                   # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop

 28: f8227805  str  x5, [x0, x2, lsl #3] # a[j] = a[j-1]
 2c: f1000442  subs x2, x2, #0x1         # j--
 30: 54ffff41  b.ne 18                   # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
 34: f8227804  str  x4, [x0, x2, lsl #3] # a[j] = x
 38: 91000463  add  x3, x3, #0x1         # i++
 3c: 17fffff2  b    4                    # กระโดดไปที่ Outer Loop
```

รูปที่ 10.11: โค้ด ARM-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 โปรแกรมภาษาแอสเซมบลี ARM-64 แตกต่างจากภาษาแอสเซมบลี ARM-32 ในรูปที่ 2.5

บันทึกและเรียกคืนรีจิสเตอร์จากสแต็ก (stack) เนื่องจากตัวนับโปรแกรม (PC) ไม่ใช่รีจิสเตอร์ จึงใช้คำสั่งคืนค่าแยกต่างหาก อันที่จริง โค้ดดูคล้ายโค้ด RV64I หรือ x86-64 มากกว่าโค้ด ARM-32

```
# MIPS-64 (24 คำสั่ง, 96 bytes)
# a1 คือ n, a3 เป็นตัวชี้ไปที่ a[0], v0 คือ j, v1 คือ i, t0 คือ x
  0: 64860008 daddiu a2,a0,8   # (daddiu vs addiu, 8 vs 4) a2 เป็นตัวชี้ไปที่ a[i]
  4: 24030001 li     v1,1      # i = 1
Outer Loop:
  8: 0065102b sltu   v0,v1,a1  # set on i < n
  c: 14400003 bnez   v0,1c     # ถ้า i < n กระโดดไปที่ Continue Outer Loop
 10: 00c03825 move   a3,a2     # a3 เป็นตัวชี้ไปที่ a[j] (slot filled)
 14: 03e00008 jr     ra        # ออกจากฟังก์ชัน
 18: 00000000 nop              # branch delay slot unfilled
Continue Outer Loop:
 1c: dcc80000 ld     a4,0(a2)  # (ld vs lw) x = a[i]
 20: 00601025 move   v0,v1     # j = i
Inner Loop:
 24: dce9fff8 ld     a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
 28: 0109502a slt    a6,a4,a5  # (no load delay slot) set a[i] < a[j-1]
 2c: 11400005 beqz   a6,44     # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
 30: 00000000 nop              # branch delay slot unfilled
 34: 6442ffff daddiu v0,v0,-1  # (daddiu vs addiu) j--
 38: fce90000 sd     a5,0(a3)  # (sd vs sw, a5 vs t1) a[j] = a[j-1]
 3c: 1440fff9 bnez   v0,24     # ถ้า j != 0 กระโดดไปที่ Inner Loop (next slot filled)
 40: 64e7fff8 daddiu a3,a3,-8  # (daddiu vs addiu, 8 vs 4) ลด a3 ตัวชี้ไปที่ a[j]
Exit Inner Loop:
 44: 000210f8 dsll   v0,v0,0x3 # (dsll vs sll)
 48: 0082102d daddu  v0,a0,v0  # (daddu vs addu) v0 now byte address of a[j]
 4c: fc480000 sd     a4,0(v0)  # (sd vs sw) a[j] = x
 50: 64630001 daddiu v1,v1,1   # (daddiu vs addiu) i++
 54: 1000ffec b      8         # กระโดดไปที่ Outer Loop (next delay slot filled)
 58: 64c60008 daddiu a2,a2,8   # (daddiu vs addiu, 8 vs 4) เพิ่ม a2 ตัวชี้ไปที่ a[i]
 5c: 00000000 nop              # Unncessary(?)
```

รูปที่ 10.12: โค้ด MIPS-64 สำหรับการเรียงลำดับการแทรกในรูปที่ 2.5 โปรแกรมภาษาแอสเซมบลี MIPS-64 มีความแตกต่างหลายอย่างจากภาษาแอสเซมบลี MIPS-32 ในรูปที่ 2.10 หน้าที่ 36 บทที่ 2 ประการแรก คำสั่งส่วนใหญ่สำหรับข้อมูล 64 บิตจะเติมตัวอักษร “d” นำหน้าชื่อ เช่น `daddiu, daddu, dsll` เช่นเดียวกับรูปที่ 10.10 คำสั่ง 3 คำสั่งเปลี่ยนค่าคงที่จาก 4 เป็น 8 เนื่องจากขนาดของข้อมูลเพิ่มจาก 4 เป็น 8 ไบต์ เช่นเดียวกับ RV64I ความกว้างที่เพิ่มขึ้นเปลี่ยนค่าคำสั่งโหลดเวิร์ด 2 คำสั่ง (`lw`) เป็นโหลดดับเบิลเวิร์ด (`ld`) และคำสั่งบันทึกเวิร์ด 2 คำสั่ง (`sw`) เป็นบันทึกดับเบิลเวิร์ด (`sd`). สุดท้าย MIPS-64 ไม่มีช่องหน่วงการโหลด จาก MIPS-32 ซึ่งทำให้ไปป์ไลน์หยุดชะงักเมื่อมีการอ่านหลังเขียน

```
# x86-64 (15 คำสั่ง, 46 bytes)
# rax คือ j, rcx คือ x, rdx คือ i, rsi คือ n, rdi เป็นตัวชี้ไปที่ a[0]
  0: ba 01 00 00 00 mov edx,0x1
Outer Loop:
  5: 48 39 f2       cmp rdx,rsi             # เปรียบเทียบ i vs. n
  8: 73 23          jae 2d <Exit Loop>      # ถ้า i >= n กระโดดไปที่ Exit Outer Loop
  a: 48 8b 0c d7    mov rcx,[rdi+rdx*8]     # x = a[i]
  e: 48 89 d0       mov rax,rdx             # j = i
Inner Loop:
 11: 4c 8b 44 c7 f8 mov r8,[rdi+rax*8-0x8] # r8 = a[j-1]
 16: 49 39 c8       cmp r8,rcx              # เปรียบเทียบ a[j-1] vs. x
 19: 7e 09          jle 24 <Exit Loop>      # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit InnerLoop
 1b: 4c 89 04 c7    mov [rdi+rax*8],r8      # a[j] = a[j-1]
 1f: 48 ff c8       dec rax                 # j--
 22: 75 ed          jne 11 <Inner Loop>     # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit InnerLoop:
 24: 48 89 0c c7    mov [rdi+rax*8],rcx     # a[j] = x
 28: 48 ff c2       inc rdx                 # i++
 2b: eb d8          jmp 5 <Outer Loop>      # กระโดดไปที่ Outer Loop
Exit Outer Loop:
 2d: c3             ret                     # ออกจากฟังก์ชัน
```

รูปที่ 10.13: โค้ด x86-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 โปรแกรมภาษาแอสเซมบลี x86-64 แตกต่างจากภาษาแอสเซมบลี x86-32 ในรูปที่ 2.11 หน้าที่ 37 บทที่ 2 ประการแรก ต่างจาก RV64I รีจิสเตอร์ที่มีความกว้างมากขึ้นมีชื่อต่างกัน เช่น `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` ประการที่สอง เนื่องจาก x86-64 เพิ่มรีจิสเตอร์อีก 8 ตัว จึงมีรีจิสเตอร์เพียงพอที่จะเก็บตัวแปรทั้งหมดไว้ในรีจิสเตอร์แทนในหน่วยความจำ ประการที่สาม คำสั่ง x86-64 มีความยาวมากกว่าคำสั่ง x86-32 เนื่องจากหลายคำสั่งต้องเติมรหัส 8 บิตหรือ 16 บิตเพื่อให้เข้ากับพื้นที่รหัสคำสั่ง (opcode space) ใหม่ ตัวอย่างเช่น การเพิ่มหรือลดรีจิสเตอร์ (`inc`, `dec`) ใช้ 1 ไบต์ แต่ใช้ 3 ไบต์ใน x86-32 ดังนั้น แม้ว่าจะใช้คำสั่งน้อยลงมาก ขนาดโค้ด x86-64 สำหรับการเรียงลำดับการแทรกเกือบจะเท่ากับ x86-32: 45 ไบต์ เทียบกับ 46 ไบต์)

11 RV32/64 สถาปัตยกรรมสิทธิพิเศษ

เอ็ดเกอร์ ไวบี ไดจ์คสตรา (ค.ศ. 1930–2020) ผู้ได้รับรางวัล Turing Award ปี 1972 สำหรับการมีส่วนร่วมพื้นฐานในการพัฒนาภาษาโปรแกรม



ความเรียบง่ายเป็นสิ่งจำเป็นสำหรับความน่าเชื่อถือ

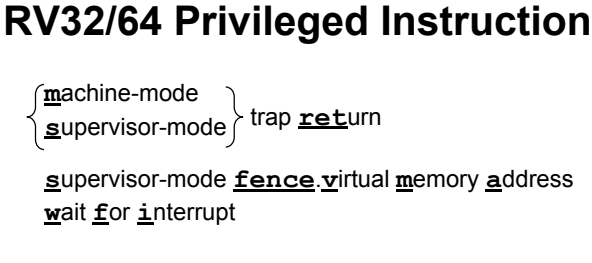
—เอ็ดเกอร์ ไวบี ไดจ์คสตรา

11.1 บทนำ

จนถึงตอนนี้ หนังสือเล่มนี้เน้นการสนับสนุนของ RISC-V สำหรับการคำนวณทั่วไป: คำสั่งทั้งหมดที่เราได้แนะนำสามารถใช้ได้ใน *โหมดผู้ใช้* (user mode) ซึ่งเป็นที่ที่โค้ดแอปพลิเคชันมักทำงาน บทนี้จะแนะนำโหมด*สิทธิพิเศษ* (privilege modes) ใหม่สองโหมด: *โหมดเครื่องจักร (machine mode)* ซึ่งรันโค้ดที่มีความน่าเชื่อถือสูงสุด และ *โหมดซูเปอร์ไวเซอร์ (supervisor mode)* ซึ่งให้การสนับสนุนระบบปฏิบัติการ เช่น Linux, FreeBSD และ Windows โหมดใหม่ทั้งสองนี้มีสิทธิพิเศษมากกว่าโหมดผู้ใช้ จึงเป็นที่มาของชื่อบทนี้

โหมดที่มีสิทธิพิเศษมากกว่า มักสามารถเข้าถึงคุณสมบัติทั้งหมดของโหมดที่มีสิทธิพิเศษน้อยกว่า และยังเพิ่มฟังก์ชันการทำงานเพิ่มเติมที่โหมดที่มีสิทธิพิเศษน้อยกว่าไม่สามารถเข้าถึงได้ เช่น ความสามารถในการจัดการการขัดจังหวะ (interrupts) และการดำเนินการ I/O โดยทั่วไป โปรเซสเซอร์จะใช้เวลาส่วนใหญ่ในการรันในโหมดที่มีสิทธิพิเศษน้อยที่สุดการขัดจังหวะและข้อยกเว้นจะโอนการควบคุมไปยังโหมดที่มีสิทธิพิเศษมากกว่า

รันไทม์ของระบบฝังตัวและระบบปฏิบัติการใช้คุณสมบัติของโหมดใหม่เหล่านี้ เพื่อตอบสนองต่อเหตุการณ์ภายนอก เช่น การมาถึงของแพ็กเก็ตเครือข่าย, เพื่อสนับสนุนการทำงานหลายงานพร้อมกัน (multitasking) และการป้องกันระหว่างงาน และเพื่อ



รูปที่ 11.1: ไดอะแกรมของคำสั่งสิทธิพิเศษของ RISC-V

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0001000		00010	00000	000	00000	1110011	R sret
0011000		00010	00000	000	00000	1110011	R mret
0001000		00101	00000	000	00000	1110011	R wfi
0001001		rs2	rs1	000	00000	1110011	R sfence.vma

รูปที่ **11.2**: รูปแบบคำสั่งสิทธิพิเศษของ RISC-V, รหัสดำเนินการ, ประเภทรูปแบบคำสั่ง และชื่อคำสั่ง (ตารางที่ 6.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

นามธรรมและจำลองคุณสมบัติของฮาร์ดแวร์ ด้วยความกว้างของหัวข้อเหล่านี้ คู่มือสำหรับโปรแกรมเมอร์ที่ครอบคลุมคงต้องใช้หนังสือเล่มใหม่ทั้งเล่ม แต่บทนี้มุ่งเน้นไปที่จุดเด่นของฟีเจอร์ RISC-V โปรแกรมเมอร์ที่ไม่สนใจรันไทม์ของระบบฝังตัวและระบบปฏิบัติการสามารถข้าม หรืออ่านผ่านบทนี้ได้


ความเรียบง่าย

รูปที่ 11.1 เป็นการนำเสนอแบบกราฟิกของคำสั่งสิทธิพิเศษของ RISC-V และ รูปที่ 11.2 ระบุรหัสดำเนินการของคำสั่งเหล่านี้ จะเห็นได้ว่า สถาปัตยกรรมสิทธิพิเศษเพิ่มคำสั่งเพียงไม่กี่คำสั่ง แต่กลับเพิ่มรีจิสเตอร์ควบคุมและสถานะ (CSRs) ใหม่หลายตัวเพื่อเปิดฟังก์ชันการทำงานเพิ่มเติม

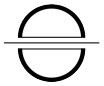
บทนี้จะอธิบายเกี่ยวกับสถาปัตยกรรมสิทธิพิเศษของ RV32 และ RV64 พร้อมกัน แนวคิดบางอย่างแตกต่างกันเพียงขนาดของรีจิสเตอร์จำนวนเต็ม ดังนั้นเพื่อให้คำอธิบายกระชับ เราใช้คำว่า XLEN เพื่ออ้างถึงความกว้างของรีจิสเตอร์จำนวนเต็มในหน่วยบิต โดย XLEN มีค่าเป็น 32 สำหรับ RV32 หรือ 64 สำหรับ RV64

11.2 โหมดเครื่องจักรสำหรับระบบฝังตัวอย่างง่าย

โหมดเครื่องจักร หรือย่อว่า M-โหมด เป็นโหมดที่มีสิทธิพิเศษสูงสุดที่ hart (hardware thread) ของ RISC-V สามารถทำงานได้ Harts ที่ทำงานใน M-mode สามารถเข้าถึงหน่วยความจำ, อินพุต/เอาต์พุต (I/O), และคุณสมบัติระบบระดับต่ำที่จำเป็นสำหรับการบูต และกำหนดค่าระบบได้อย่างเต็มที่ ดังนั้น มันเป็นโหมดสิทธิพิเศษเพียงโหมดเดียวที่โปรเซสเซอร์ RISC-V มาตรฐานทั้งหมดต้องมี และในความเป็นจริง ไมโครคอนโทรลเลอร์ RISC-V อย่างง่ายรองรับ เฉพาะ M-mode เท่านั้น ระบบดังกล่าวเป็นจุดเน้นของส่วนนี้

Hart* เป็น คำ ย่อ ของ *hardware thread เราใช้คำนี้ เพื่อแยกความแตกต่างจาก software threads ซึ่งโปรแกรมเมอร์ ส่วนใหญ่ คุ้น เคย software threads ถูก แบ่ง เวลา (time-multiplexed) บน hart แกนประมวลผลส่วนใหญ่มีเพียง hart เดียว

คุณสมบัติที่สำคัญที่สุดของโหมดเครื่องจักร คือความสามารถในการดักจับและจัดการข้อยกเว้น (exceptions): เหตุการณ์รันไทม์ที่ผิดปกติ RISC-V แบ่งข้อยกเว้นออกเป็นสองประเภท: *ข้อยกเว้นซิงโครนัส* (synchronous exceptions) เกิดขึ้นจากการทำงานของคำสั่ง เช่น การเข้าถึงที่อยู่หน่วยความจำที่ไม่ถูกต้อง หรือการทำงานของคำสั่งที่มีรหัสดำเนินการไม่ถูกต้อง *การขัดจังหวะ* (Interrupts) เป็นเหตุการณ์ภายนอกที่ไม่ซิงโครนัสกับลำดับคำสั่ง เช่น การคลิกปุ่มเมาส์ ข้อยกเว้นใน RISC-V เป็นแบบ *'แม่นยำ (precise)*: คำสั่งทั้งหมดก่อนหน้าข้อยกเว้นจะทำงานสมบูรณ์ และไม่มีคำสั่งที่ตามมาจะเริ่มทำงาน รูปที่ 11.3 ระบุสาเหตุของข้อยกเว้นมาตรฐาน


แยกสถาปัตย์ กับนำไปใช้

ข้อยกเว้นแบบซิงโครนัสห้าประเภทที่สามารถเกิดขึ้นได้ระหว่างการทำงานในโหมด

ข้อ ยกเว้น ที่ เกี่ยว กับ ที่ อยู่ คำ สั่ง ที่ไม่ เรียง ตัว ไม่ สามารถ เกิด ขึ้น กับ ส่วน ขยาย C เพราะไม่สามารถกระโดดไปยังที่อยู่ที่เป็น

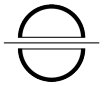
Interrupt/$\overline{\text{Exception}}$ `mcause`[XLEN-1]	Exception Code `mcause`[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

รูปที่ 11.3: สาเหตุของข้อยกเว้นและขัดจังหวะใน RISC-V บิตที่มีนัยสำคัญที่สุดของ `mcause` ถูกตั้งค่าเป็น 1 สำหรับขัดจังหวะ หรือ 0 สำหรับข้อยกเว้นซิงโครนัส และบิตที่มีนัยสำคัญน้อยที่สุดระบุประเภทของการขัดจังหวะหรือข้อยกเว้น และการขัดจังหวะของโหมดซุปเปอร์ไวเซอร์ และข้อยกเว้นจากข้อผิดพลาดของเพจ (page-fault exceptions) จะเกิดขึ้นได้เฉพาะเมื่อมีการใช้งานโหมดซูเปอร์ไวเซอร์ (ดูหัวข้อ 11.5) (ตารางที่ 3.6 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

เครื่องจักร (M-mode):

- *ข้อยกเว้นการเข้าถึงผิดพลาด (Access fault exceptions)* เกิดขึ้นเมื่อที่อยู่หน่วยความจำกายภาพไม่รองรับประเภทการเข้าถึง—เช่น การพยายามบันทึกข้อมูลลงใน ROM
- *ข้อยกเว้นจากจุดพัก* เกิดจากการทำงานคำสั่ง `ebreak` หรือเมื่อที่อยู่หรือข้อมูลตรงกับทริกเกอร์การดีบัก
- *ข้อยกเว้นจากการเรียกสภาพแวดล้อม* เกิดจากการทำงานคำสั่ง `ecall`
- *ข้อยกเว้นคำสั่งที่ไม่ถูกต้อง* เกิดจากการถอดรหัสดำเนินการที่ไม่ถูกต้อง
- *ข้อยกเว้นที่อยู่ที่ไม่เรียงตัว* เกิดขึ้นเมื่อที่อยู่ที่มีผลไม่สามารถหารลงตัวด้วยขนาดการเข้าถึง (Access size)—เช่น `amoadd.w` กับตำแหน่งที่อยู่ `0x12`

หากคุณจำได้ในบทที่ 2 ที่ระบุว่า RISC-V อนุญาตให้มีการโหลดและบันทึกที่ไม่เรียงตัว คุณอาจสงสัยว่า ทำไมข้อยกเว้นตำแหน่งที่อยู่โหลดและจัดเก็บที่[ไม่เรียงตัว]ได้รับอนุญาต] คุณอาจสงสัยว่าทำไมข้อยกเว้นที่เกี่ยวกับที่อยู่โหลดและบันทึกที่ไม่เรียงตัวจึงปรากฏในรูปที่ 11.3 มีสองเหตุผลสองประการ ประการแรก การดำเนินการหน่วยความจำอะตอมิก ในบทที่ 6 ต้องการที่อยู่ที่เรียงตามธรรมชาติ (naturally aligned addresses) ประการที่สอง ผู้พัฒนาบางรายเลือกที่จะไม่รวมการสนับสนุนในฮาร์ดแวร์สำหรับการโหลดและบันทึกที่ไม่เรียงตัวทั่วไป เนื่องจากเป็นคุณสมบัติที่ยากต่อการพัฒนาและใช้งานไม่บ่อย โปรเซสเซอร์ที่ไม่มีฮาร์ดแวร์นี้จะพึ่งพาตัวจัดการข้อยกเว้น (exception handler) เพื่อดักจับและจำลอง (emulate) การโหลดและบันทึกที่ไม่เรียงตัวในซอฟต์แวร์ โดยใช้ลำดับของการโหลดและบันทึกขนาดเล็กที่เรียงตัว โค้ดแอปพลิเคชันไม่รู้ตัวถึงความแตกต่าง: การเข้าถึงหน่วยความจำที่ไม่เรียงตัวทำงานได้ตามที่คาดไว้ แม้ว่าจะช้าก็ตาม ในขณะที่ฮาร์ดแวร์ยังคงเรียบง่าย หรืออีกทางเลือกหนึ่ง โปรเซสเซอร์ที่มีประสิทธิภาพสูงกว่าสามารถพัฒนาการโหลดและบันทึกที่ไม่เรียงตัวในฮาร์ดแวร์ได้ ความยืดหยุ่นในการพัฒนานี้เกิดจากการตัดสินใจของ RISC-V ที่อนุญาตให้มีการโหลดและบันทึกที่ไม่เรียงตัวโดยใช้รหัสคำสั่ง (opcodes) การโหลดและบันทึกทั่วไป ตามแนวทางในบทที่ 1 ที่แยกสถาปัตยกรรมออกจากการพัฒนา



มีแหล่งที่มาของการขัดจังหวะ (interrupts) มาตรฐานสามประเภท: การขัดจังหวะซอฟต์แวร์ (Software interrupts): เกิดจากการบันทึกข้อมูลลงในรีจิสเตอร์ที่แมปกับหน่วยความจำ (memory-mapped register) และมักใช้โดย hart หนึ่งเพื่อขัดจังหวะ hart อื่น ซึ่งเป็นกลไกที่สถาปัตยกรรมอื่นเรียกว่า การขัดจังหวะระหว่างโปรเซสเซอร์ (**interprocessor interrupt**) การขัดจังหวะตัวจับเวลา (Timer interrupts): เกิดขึ้นเมื่อตัวเปรียบเทียบเวลาของ hart ซึ่งเป็นรีจิสเตอร์ที่แมปกับหน่วยความจำชื่อ `mtimecmp` มีค่าเท่ากับหรือมากกว่าตัวนับเวลาจริงชื่อ `mtime` การขัดจังหวะภายนอก (External interrupts): เกิดจากตัวควบคุมการขัดจังหวะระดับแพลตฟอร์ม ซึ่งอุปกรณ์ภายนอกส่วนใหญ่เชื่อมต่ออยู่ เนื่องจากแพลตฟอร์มฮาร์ดแวร์ที่แตกต่างกันมีแผนที่หน่วยความจำ (memory maps) และความต้องการคุณสมบัติของตัวควบคุมการขัดจังหวะที่แตกต่างกัน

XLEN-1	XLEN-2	23	22	21	20	19	18	17	
SD	*Reserved*		TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-24		1	1	1	1	1	1	

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	*Res.*	SPP	MPIE	*Res.*	SPIE	*Res.*	MIE	*Res.*	SIE	*Res.*
2	2	2	2	1	1	1	1	1	1	1	1	1

รูปที่ 11.4: `mstatus` CSR ในโปรเซสเซอร์อย่างง่ายที่มีเฉพาะโหมดเครื่องจักรและไม่มีส่วนขยาย F (Floating-Point) และ V (Vector) ฟิลด์ที่มีอยู่จะมีเพียงตัวเปิดใช้งานการขัดจังหวะทั่วไป (global interrupt enable) MIE และ MPIE ซึ่งหลังจากเกิดข้อยกเว้นจะเก็บค่าเดิมของ MIE ค่า XLEN คือ 32 สำหรับ RV32 หรือ 64 สำหรับ RV64 รูปที่ 3.7 ใน [Waterman and Asanović 2017] เป็นที่มาของรูปนี้ ดูหัวข้อ 3.1 ของเอกสารนั้นสำหรับคำอธิบายของฟิลด์อื่นๆ

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		MEIP	*Res.*	SEIP	*Res.*	MTIP	*Res.*	STIP	*Res.*	MSIP	*Res.*	SSIP	*Res.*
Reserved		MEIE	*Res.*	SEIE	*Res.*	MTIE	*Res.*	STIE	*Res.*	MSIE	*Res.*	SSIE	*Res.*
XLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

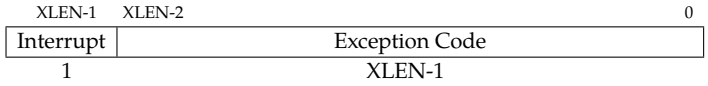
รูปที่ 11.5: รีจิสเตอร์ควบคุมและสถานะการขัดจังหวะของโหมดเครื่องจักร เป็นรีจิสเตอร์อ่าน/เขียนขนาด XLEN บิต ที่เก็บการขัดจังหวะที่รอดำเนินการ (pending interrupts, `mip`) และบิตเปิดใช้งานการขัดจังหวะ (pending interrupts, `mie`) เฉพาะบิตที่สอดคล้องกับการขัดจังหวะซอฟต์แวร์ระดับต่ำกว่า (SSIP), การขัดจังหวะตัวจับเวลา (STIP) และการขัดจังหวะภายนอก (SEIP) ใน `mip` เท่านั้น ที่สามารถเขียนได้ผ่านที่อยู่ CSR นี้ ส่วนบิตที่เหลือเป็นแบบอ่านอย่างเดียว

กลไกสำหรับการก่อให้เกิดและล้างการขัดจังหวะเหล่านี้จึงแตกต่างกันไปตามแพลตฟอร์ม อย่างไรก็ตาม สิ่งที่ คงที่ ในทุกระบบ RISC-V คือวิธีการจัดการข้อยกเว้นและการปิดกั้นการขัดจังหวะ (masking interrupts) ซึ่งเป็นหัวข้อของส่วนถัดไป

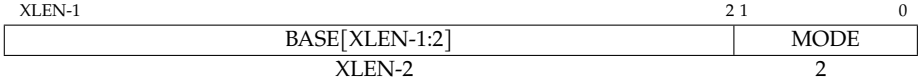
11.3 การจัดการข้อยกเว้นในโหมดเครื่องจักร

รีจิสเตอร์ควบคุมและสถานะ (CSRs) 8 ตัว ที่สำคัญต่อการจัดการข้อยกเว้นในโหมดเครื่องจักร:

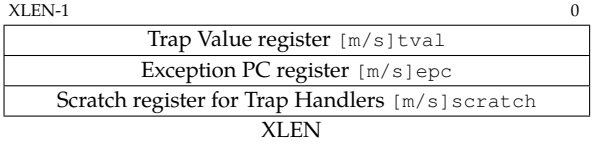
- `mstatus`, *Machine Status* สถานะโหมดเครื่องจักร เก็บการเปิดใช้งานการขัดจังหวะทั่วไปพร้อมกับสถานะอื่นๆ อีกมากมาย ตามที่แสดงในรูปที่ 11.4
- `mcause`, *Machine Exception Cause* สาเหตุข้อยกเว้นโหมดเครื่องจักรระบุว่ามีข้อยกเว้นใดเกิดขึ้น (รูปที่ 11.6)
- `mie`, *Machine Interrupt Enable* ตัวเปิดใช้งานการขัดจังหวะโหมดเครื่องจักร ระบุว่าการขัดจังหวะใดที่โปรเซสเซอร์สามารถรับได้และต้องละเว้น (รูปที่ 11.5)



รูปที่ 11.6: รีจิสเตอร์สาเหตุของโหมดเครื่องจักรและโหมดหัวหน้างาน (`mcause` และ `scause`) เมื่อเกิดการดักจับ (trap) รีจิสเตอร์ CSR จะถูกตั้งค่าเป็น 1 หากการดักจับเกิดจากการขัดจังหวะ บิต Interrupt จะถูกตั้งค่าหากการดักจับเกิดจากการขัดจังหวะ ฟิลด์รหัสข้อยกเว้น (Exception Code) เก็บรหัสที่ระบุข้อยกเว้นล่าสุด (last exception) รูปที่ 11.3 แสดงการ่แมปค่ารหัสไปยังสาเหตุของการดักจับ



รูปที่ 11.7: รีจิสเตอร์ที่อยู่ฐานของเวกเตอร์การดักจับในโหมดเครื่องจักรและโหมดหัวหน้างาน (`mtvec` และ `stvec`) เป็นรีจิสเตอร์อ่าน/เขียนขนาด XLEN บิต ที่เก็บการกำหนดค่าเวกเตอร์การดักจับ ซึ่งประกอบด้วยที่อยู่ฐานของเวกเตอร์ (BASE) และโหมดเวกเตอร์ (MODE) ค่าในฟิลด์ BASE ต้องเรียงตัวบนขอบเขต 4 ไบต์เสมอ (4-byte boundary) MODE = 0 ข้อยกเว้นทั้งหมดตั้งค่าตัวนับโปรแกรม (PC) ไปที่ BASE MODE = 1 ตั้งค่า PC เป็น $(BASE + (4 \times cause))$ สำหรับการขัดจังหวะแบบอะซิงโครนัส



รูปที่ 11.8: รีจิสเตอร์ CSR ที่เกี่ยวข้องกับข้อยกเว้นและการขัดจังหวะ รีจิสเตอร์ค่าเหตุการณ์ดักจับ (`mtval` และ `stval`) เก็บข้อมูลการเพิ่มเติมของการดักจับ เช่น ตำแหน่งที่อยู่ที่ผิดพลาดหรือคำสั่งที่ไม่ถูกต้อง รีจิสเตอร์ตัวนับโปรแกรมของข้อยกเว้น (`mepc` และ `sepc`) ชี้ไปที่คำสั่งที่ทำให้เกิดข้อยกเว้น ตัวรีจิสเตอร์ชั่วคราว (`mscratch` และ `sscratch`) ให้ตัวจัดการการดักจับ (trap handlers) มีรีจิสเตอร์หนึ่งตัวสำหรับใช้งาน

Encoding	Name	Abbreviation
00	User	U
01	Supervisor	S
10	Hypervisor	H
11	Machine	M

รูปที่ 11.9: ระดับสิทธิพิเศษของ RISC-V และการเข้ารหัส

- mip, *Machine Interrupt Pending* การขัดจังหวะรอดำเนินการระบุว่ามีการขัดจังหวะใดที่รอดำเนินการ (pending) (รูปที่ 11.5
- mtvec, *Machine Trap Vector* เวกเตอร์การดักจับเครื่องจักรเก็บที่อยู่ที่โปรเซสเซอร์กระโดดไปเมื่อเกิดข้อยกเว้น รูปที่ 11.7)
- mtval, *Machine Trap Value* ค่าการดักจับโหมดเครื่องจักรเก็บข้อมูลการดักจับเพิ่มเติมของการดักจับ เช่น ที่อยู่ที่ข้อผิดพลาดสำหรับข้อยกเว้นที่เกี่ยวกับที่อยู่, คำสั่งสำหรับข้อยกเว้นคำสั่งไม่ถูกต้อง, และศูนย์สำหรับข้อยกเว้นอื่นๆ (รูปที่ 11.8)
- mepc, *Machine Exception PC* ตัวนับโปรแกรมข้อยกเว้นโหมดเครื่องจักรชี้ไปยังคำสั่งที่เกิดข้อยกเว้น (รูปที่ 11.8)
- mscratch, *Machine Scratch* รีจิสเตอร์ชั่วคราวโหมดเครื่องจักรเก็บข้อมูลหนึ่งเวิร์ดสำหรับการจัดเก็บชั่วคราวของตัวจัดการการดักจับ (รูปที่ 11.8)

เมื่อทำงานในโหมดเครื่องจักร (M-mode) การขัดจังหวะจะเกิดขึ้นได้เฉพาะเมื่อบิตตัวเปิดใช้งานการขัดจังหวะทั่วไป mstatus.MIE ถูกตั้งค่าเป็น 1 นอกจากนี้ การขัดจังหวะแต่ละประเภทมีบิตเปิดใช้งานของตัวเองในรีจิสเตอร์ CSR mie ตำแหน่งบิตใน mie สอดคล้องกับรหัสการขัดจังหวะ ในรูปที่ 11.3 เช่น mie[7] สอดคล้องกับการขัดจังหวะตัวจับเวลาโหมดเครื่องจักร รีจิสเตอร์ CSR mip มีรูปแบบเดียวกันและระบุว่าการขัดจังหวะใดที่กำลังรอดำเนินการ การรวมทั้งสามรีจิสเตอร์ CSR (mstatus, mie, mip) การขัดจังหวะตัวจับเวลาเครื่องจักรจะเกิดขึ้นได้หาก mstatus.MIE=1, mie[7]=1 และ mip[7]=1

RISC-V รองรับ การขัดจังหวะแบบเวกเตอร์ซึ่งโปรเซสเซอร์จะกระโดดไปยังที่อยู่เฉพาะของการขัดจังหวะนั้นๆ แทนที่จะไปยังจุดเริ่มต้นเดียว (single entry point) การกำหนดที่อยู่นี้ช่วยลดความจำเป็นในการอ่านและถอดรหัส mcause ทำให้การจัดการการขัดจังหวะรวดเร็วขึ้น การตั้งค่า mtvec[0] เป็น 1 จะเปิดใช้งานคุณลักษณะนี้ โดยการขัดจังหวะที่มีสาเหตุ x จะตั้งค่า PC ไปที่ (mtvec-1+4x) แทนที่ mtvec ทั่วไป

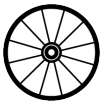
เมื่อ hart เกิดข้อยกเว้น ฮาร์ดแวร์จะทำการเปลี่ยนสถานะแบบอะตอมิก (atomic state transitions) ดังนี้:

- PC ของคำสั่งที่ทำให้เกิดข้อยกเว้นถูกเก็บรักษาไว้ใน mepc และ PC จะถูกตั้งค่าไปที่ mtvec (สำหรับข้อยกเว้นแบบซิงโครนัส mepc ชี้ไปยังคำสั่งที่ทำให้เกิดข้อยกเว้น สำหรับการขัดจังหวะ ชี้ไปยังตำแหน่งที่การดำเนินการควรเริ่มต่อหลังจัดการการขัดจังหวะ)
- mcause ถูกตั้งค่าเป็นรหัสสาเหตุของข้อยกเว้น ตามที่เข้ารหัส ในรูป 11.3 และ mtval ถูกตั้งค่าเป็นที่อยู่ที่ผิดพลาดหรือข้อมูลคำเฉพาะของข้อยกเว้น
- การขัดจังหวะถูกปิดใช้งานโดยการตั้งค่า MIE=0 ในรีจิสเตอร์ CSR mstatus และค่า MIE ก่อนหน้าจะถูกเก็บไว้ใน MPIE

- โหมดสิทธิพิเศษก่อนเกิดข้อยกเว้นถูกเก็บไว้ในฟิลด์ MPP ของ `mstatus` และโหมดสิทธิพิเศษจะถูกเปลี่ยนเป็นโหมดเครื่องจักร (M-mode) รูปที่ 11.9 แสดงการเข้ารหัสของฟิลด์ MPP (หากโปรเซสเซอร์รองรับเฉพาะโหมด M ขั้นตอนนี้จะถูกข้ามไปโดยปริยาย)

เพื่อหลีกเลี่ยงการเขียนทับเนื้อหาของรีจิสเตอร์จำนวนเต็ม ส่วนนำ (prologue) ของตัวจัดการการขัดจังหวะมักเริ่มต้นด้วยการสลับรีจิสเตอร์จำนวนเต็ม (เช่น a0) กับรีจิสเตอร์ CSR `mscratch` โดยทั่วไป ซอฟต์แวร์จะจัดเตรียมให้ `mscratch` เก็บตัวชี้ไปยังพื้นที่เก็บข้อมูลชั่วคราวในหน่วยความจำ ซึ่งตัวจัดการใช้เพื่อบันทึกรีจิสเตอร์จำนวนเต็มตามที่ตัวจัดการต้องการ หลังจากรันส่วนหลัก (body) ของตัวจัดการ ส่วนจบ (epilogue) จะกู้คืนรีจิสเตอร์ที่บันทึกไว้ในหน่วยความจำ จากนั้นสลับ a0 กับ `mscratch` อีกครั้ง เพื่อกู้คืนทั้งสองรีจิสเตอร์ให้เป็นสถานะก่อนเกิดข้อยกเว้น สุดท้าย ตัวจัดการจะคืนค่าด้วยคำสั่ง `mret` ซึ่งเป็นคำสั่งเฉพาะของโหมด M คำสั่ง `mret` จะตั้งค่า PC เป็น `mepc` คัดลอกฟิลด์ MPIE ของ `mstatus` ไปยัง MIE เพื่อกู้คืนการตั้งค่าการเปิดใช้งานการขัดจังหวะก่อนหน้า และตั้งค่าโหมดสิทธิพิเศษเป็นค่าที่อยู่ในฟิลด์ MPP ของ `mstatus` โดยพื้นฐานแล้วเป็นการย้อนกลับการกระทำที่อธิบายไว้ในย่อหน้าก่อนหน้า

รูปที่ 11.10 แสดงโค้ดแอสเซมบลี RISC-V สำหรับตัวจัดการการขัดจังหวะตัวจับเวลาพื้นฐานที่ปฏิบัติตามรูปแบบนี้ โดยมันเพียงเพิ่มค่าตัวเปรียบเทียบเวลาแล้วกลับไปยังงานก่อนหน้า ในขณะที่ตัวจัดการการขัดจังหวะตัวจับเวลาที่สมจริงกว่านี้ อาจเรียกตัวจัดตารางงาน (scheduler) เพื่อสลับระหว่างงาน ตัวจัดการนี้ไม่สามารถถูกขัดจังหวะล่วงหน้า (non-preemptible) จึงปิดการขัดจังหวะไว้ตลอดตัวจัดการ (handler) โดยไม่คำนึงถึงข้อจำกัดเหล่านั้น มันเป็นตัวอย่างที่สมบูรณ์ของตัวจัดการการขัดจังหวะ RISC-V บนหน้าเดียว!


ความเรียบง่าย

บางครั้งจำเป็นต้องรับการขัดจังหวะที่มีลำดับความสำคัญสูงกว่าในขณะที่กำลังประมวลผลข้อยกเว้นที่มีลำดับความสำคัญต่ำกว่า น่าเสียดายที่มีเพียงสำเนาเดียวของรีจิสเตอร์ CSR `mepc`, `mcause`, `mtval` และ `mstatus`; การขัดจังหวะครั้งที่สองจะทำลายค่าเก่าในรีจิสเตอร์เหล่านี้ ทำให้ข้อมูลสูญหายหากไม่มีซอฟต์แวร์ช่วยเหลือเพิ่มเติม ตัวจัดการการขัดจังหวะที่สามารถถูกขัดจังหวะล่วงหน้า (preemptible) สามารถบันทึกรีจิสเตอร์เหล่านี้ลงในสแต็กในหน่วยความจำก่อนเปิดใช้งานการขัดจังหวะจากนั้น ก่อนออกจากตัวจัดการ ให้ปิดการขัดจังหวะและกู้คืนรีจิสเตอร์จากสแต็ก


การเขียนโปรแกรม

นอกจากคำสั่ง `mret` ที่แนะนำไปแล้ว โหมดเครื่องจักรยังมีคำสั่งเพิ่มเติมเพียงคำสั่งเดียวคือคำสั่ง `wfi` (*Wait For Interrupt*) คำสั่ง `wfi` แจ้งให้โปรเซสเซอร์ทราบว่าไม่มีงานที่มีประโยชน์ให้ทำ จึงควรเข้าสู่โหมดพลังงานต่ำ (lower-power mode) จนกว่าการขัดจังหวะที่เปิดใช้งานรอดำเนินการ นั่นคือ (`mie` & `mip`)≠0 โปรเซสเซอร์ RISC-V อาจพัฒนาคำสั่งนี้ในรูปแบบต่าง ๆ เช่น การหยุดนาฬิกา จนกว่าการขัดจังหวะจะรอดำเนินการหรือบางตัวเพียงแค่ดำเนินการเป็น `nop` ดังนั้น `wfi` มักใช้ภายในลูป (loop)

```
# บันทึก รีจิสเตอร์
csrrw a0, mscratch, a0  # บันทึก a0; set a0 = &ที่จัดเก็บชั่วคราว
sw a1, 0(a0)            # บันทึก a1
sw a2, 4(a0)            # บันทึก a2
sw a3, 8(a0)            # บันทึก a3
sw a4, 12(a0)           # บันทึก a4

# ถอดรหัสสาเหตุขัดจังหวะ
csrr a1, mcause         # อ่านสาเหตุข้อยกเว้น
bgez a1, exception      # กระโดดถ้าไม่ขัดจังหวะ
andi a1, a1, 0x3f       # แยกสาเหตุข้อยกเว้น
li a2, 7                # a2 = ตรวจสอบเวลาเหตุข้อยกเว้น
bne a1, a2, otherInt    # กระโดดถ้าไม่มีเวลาขัดจังหวะ

# การจัดการเวลาในการขัดจังหวะโดยการเพิ่มเวลาในการเปรียบเทียบ
la a1, mtimecmp         # a1 = &เปรียบเทียบเวลา
lw a2, 0(a1)            # ดึง 32 bits ล่าง ของตัวเปรียบเทียบ
lw a3, 4(a1)            # ดึง 32 bits บน ของตัวเปรียบเทียบ
addi a4, a2, 1000       # เพิ่ม bits ล่าง 1000 รอบ
sltu a2, a4, a2         # สร้าง ตัวทดออก (carry-out)
add a3, a3, a2          # เพิ่ม bits บน
sw a3, 4(a1)            # จัดเก็บ 32 bits บน
sw a4, 0(a1)            # จัดเก็บ 32 bits ล่าง

# คืนค่ารีจิสเตอร์และส่งคืน
lw a4, 12(a0)           # กู้คืน a4
lw a3, 8(a0)            # กู้คืน a3
lw a2, 4(a0)            # กู้คืน a2
lw a1, 0(a0)            # กู้คืน a1
csrrw a0, mscratch, a0  # กู้คืน a0; mscratch = &ที่จัดเก็บชั่วคราว
mret                    # ออกจาก handler
```

รูปที่ 11.10: โค้ด RISC-V สำหรับตัวจัดการการขัดจังหวะตัวจับเวลาแบบง่าย โค้ดนี้สมมติว่า การขัดจังหวะถูกเปิดใช้งานทั่วโลกโดยการตั้งค่า `mstatus`.MIE; การขัดจังหวะตัวจับเวลาถูกเปิดใช้งานโดยการตั้งค่า `mie`[7]; รีจิสเตอร์ CSR `mtvec` ถูกตั้งค่าเป็นที่อยู่ของตัวจัดการนี้; และรีจิสเตอร์ CSR `mscratch` ถูกตั้งค่าเป็นที่อยู่ของบัฟเฟอร์ที่มีพื้นที่ชั่วคราวขนาด 16 ไบต์เพื่อบันทึกรีจิสเตอร์ ส่วนนำ (Prologue): บันทึกห้ารีจิสเตอร์ โดยเก็บ `a0` ใน `mscratch` และ `a1–a4` ในหน่วยความจำ จากนั้นถอดรหัสสาเหตุของข้อยกเว้น โดยตรวจสอบ `mcause`: หาก `mcause`<0 หมายถึงการขัดจังหวะ หรือหาก `mcause`≥0 หมายถึงข้อยกเว้นแบบซิงโครนัส หากเป็นการขัดจังหวะ จะตรวจสอบว่าบิตล่างของ `mcause` เท่ากับ 7 ซึ่งระบุว่าเป็นการขัดจังหวะตัวจับเวลาในโหมดโหมดเครื่องจักร หากเป็นการขัดจังหวะตัวจับเวลา จะเพิ่ม 1,000 รอบให้กับตัวเปรียบเทียบเวลา เพื่อให้การขัดจังหวะตัวจับเวลาครั้งต่อไปเกิดขึ้นในอีกประมาณ 1,000 รอบในอนาคต สุดท้าย ส่วนจบ (epilogue) กู้คืนรีจิสเตอร์ `a0–a4` และ `mscratch` แล้วคืนสู่ตำแหน่งเดิมโดยใช้คำสั่ง `mret`

■ ***รายละเอียดเพิ่มเติม: การทำงานของ*** `wfi` ***ไม่ขึ้นอยู่กับว่าการขัดจังหวะจะถูกเปิดใช้งานทั่วทั้งระบบหรือไม่***

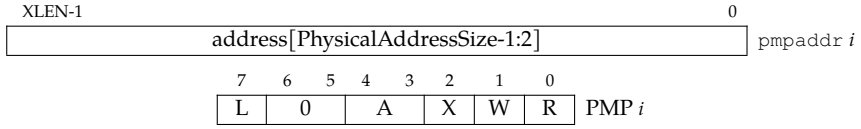
หาก wfi ถูกดำเนินการเมื่อการขัดจังหวะถูกเปิดใช้งานทั่วทั้งระบบ (mstatus.MIE=1) และมีการขัดจังหวะที่เปิดใช้งานรอดำเนินการ โปรเซสเซอร์จะกระโดดไปยังตัวจัดการข้อยกเว้น ในทางกลับกัน หาก `wfi` ถูกดำเนินการเมื่อการขัดจังหวะถูกปิดใช้งานทั่วทั้งระบบ (`mstatus.MIE=0`) และมีการขัดจังหวะที่เปิดใช้งานรอดำเนินการ โปรเซสเซอร์จะดำเนินการโค้ดที่ตามหลัง `wfi` โดยทั่วไป โค้ดนี้จะตรวจสอบรีจิสเตอร์ `mip` เพื่อตัดสินใจว่าจะทำอะไรต่อไป กลยุทธ์นี้สามารถลดความหน่วงของการขัดจังหวะเมื่อเทียบกับการกระโดดไปที่ตัวจัดการข้อยกเว้น เนื่องจากไม่จำเป็นต้องบันทึกและกู้คืนรีจิสเตอร์จำนวนเต็ม

11.4 โหมดผู้ใช้และการแยกกระบวนการในระบบฝังตัว

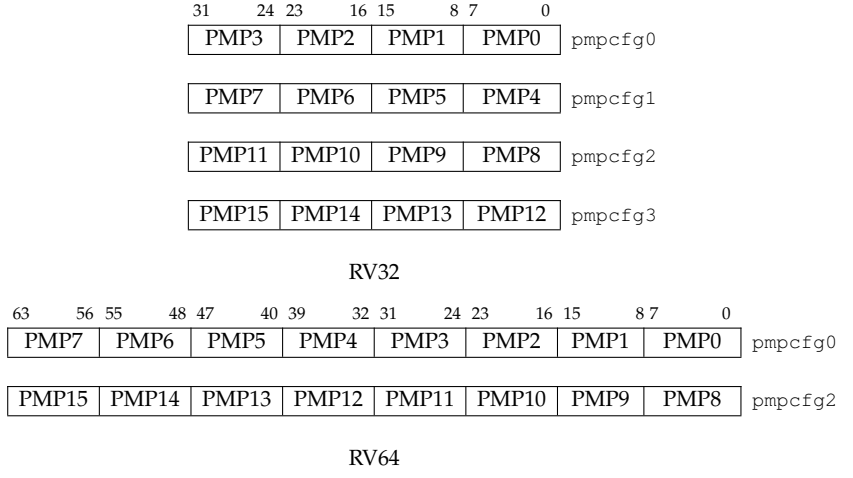
ถึงแม้ว่าโหมดเครื่องจักรจะเพียงพอสำหรับระบบฝังตัวแบบง่าย แต่เหมาะสมเฉพาะเมื่อโค้ดทั้งหมดได้รับความไว้วางใจ เนื่องจากโหมด-M สามารถเข้าถึงแพลตฟอร์มฮาร์ดแวร์ได้อย่างเต็มที่ แต่ในหลายกรณี การไว้วางใจโค้ดแอปพลิเคชันทั้งหมดเป็นไปไม่ได้ เพราะโค้ดอาจไม่เป็นที่รู้จักล่วงหน้าหรือมีขนาดใหญ่เกินกว่าที่จะพิสูจน์ความถูกต้องได้ ดังนั้น RISC-V จึงมีกลไกเพื่อปกป้องระบบจากโค้ดที่ไม่น่าเชื่อถือ (untrusted code) และปกป้องกระบวนการที่ไม่น่าเชื่อถือจากกันและกัน

โค้ดที่ไม่น่าเชื่อถือต้องถูกห้ามไม่ให้ดำเนินการคำสั่งที่มีสิทธิพิเศษ เช่น `mret` และเข้าถึงรีจิสเตอร์ CSR ที่มีสิทธิพิเศษ เช่น `mstatus` เนื่องจากสิ่งเหล่านี้จะทำให้โปรแกรมสามารถควบคุมระบบได้ การจำกัดนี้ทำได้ง่าย ๆ โดยใช้โหมดสิทธิพิเศษเพิ่มเติมคือ โหมดผู้ใช้ *User mode*(U-mode) ซึ่งปฏิเสธการเข้าถึงคุณสมบัติเหล่านี้ และสร้างข้อยกเว้นคำสั่งไม่ถูกต้อง (illegal instruction exception) เมื่อพยายามใช้คำสั่งหรือ CSR ของ M-mode ในทางอื่น U-mode และ M-mode มีพฤติกรรมคล้ายกันมาก ซอฟต์แวร์ใน M-mode สามารถเข้าสู่ U-mode ได้โดยตั้งค่าฟิลด์ `mstatus`.MPP เป็น U (ซึ่งตามที่รูปที่ 11.9 แสดง ถูกเข้ารหัสเป็น 0) แล้วดำเนินการคำสั่ง `mret` หากเกิดข้อยกเว้นใน U-mode การควบคุมจะถูกส่งกลับไปยัง M-mode

โค้ดที่ไม่น่าเชื่อถือยังต้องถูกจำกัดให้เข้าถึงเฉพาะหน่วยความจำของตัวเองโปรเซสเซอร์ที่ พัฒนา M-mode และ U-mode มี คุณสมบัติ ที่ เรียก ว่า การ ป้องกัน หน่วย ความ จำกายภาพ *Physical Memory Protection* PMP ซึ่งช่วยให้ M-mode สามารถระบุที่อยู่หน่วยความจำที่ U-mode สามารถเข้าถึงได้ PMP ประกอบด้วยรีจิสเตอร์ที่อยู่ (address registers) หลาย ตัว (โดย ปกติ คือ 8–16) และ รี จิ ส เตอร์ กำหนด ค่าที่ เกี่ยวข้อง ซึ่ง ให้หรือปฏิเสธสิทธิ์การอ่าน (read), การเขียน (write), และการดำเนินการ (execute) เมื่อโปรเซสเซอร์ใน U-mode พยายามดึงคำสั่ง (fetch an instruction) หรือดำเนินการโหลด (load) หรือบันทึก (store) ที่อยู่จะถูกเปรียบเทียบกับรีจิสเตอร์ที่อยู่ PMP ทั้งหมด หากที่อยู่นั้นมากกว่าหรือเท่ากับที่อยู่ PMP address *i* แต่น้อยกว่าที่อยู่ PMP *i*+1 รีจิสเตอร์กำหนดค่า PMP *i*+1's จะตัดสินใจว่าการเข้าถึงนั้นสามารถดำเนินการต่อได้หรือไม่ มิฉะนั้นจะเกิดข้อยกเว้นการเข้าถึง (access exception)

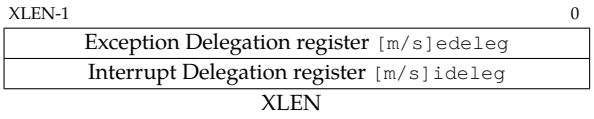


รูปที่ 11.11: รีจิสเตอร์ที่อยู่และการกำหนดค่าของ PMP รีจิสเตอร์ที่อยู่คือรีจิสเตอร์ CSR `pmpaddr` ซึ่งถูกเลื่อนขวา (right-shifted) 2 บิต และหากที่อยู่กายภาพ (physical addresses) มีความกว้างน้อยกว่า `XLEN`-2 บิต บิตด้านบนจะเป็นศูนย์ รีจิสเตอร์กำหนดค่า PMP *i* ประกอบด้วยฟิลด์บิตเพื่อควบคุมการเข้าถึงหน่วยความจำ ฟิลด์ R, W และ X ให้สิทธิ์ในการอ่าน (read), เขียน (write) และดำเนินการ (execute) ตามลำดับ ฟิลด์ A กำหนดโหมดของ PMP และฟิลด์ L (locks) ล็อก PMP และรีจิสเตอร์ที่อยู่ที่เกี่ยวข้อง



รูปที่ 11.12: ครงสร้างของการกำหนดค่า PMP ในรีจิสเตอร์ `pmpcfg` สำหรับ RV32 (ด้านบน) รีจิสเตอร์กำหนดค่าทั้ง 16 ตัวถูกบรรจุลงใน 4 CSRs สำหรับ RV64 (ด้านล่าง) รีจิสเตอร์เหล่านี้ถูกจัดเก็บในสอง CSRs ที่มีเลขคู่

รูปที่ 11.11 แสดงโครงสร้างของรีจิสเตอร์ที่อยู่และกำหนดค่าของ PMP ทั้งสองเป็นรีจิสเตอร์ควบคุมและสถานะ (CSRs) โดยรีจิสเตอร์ที่อยู่มีชื่อตั้งแต่ `pmpaddr0` ถึง `pmpaddr`*N* โดยที่ *N*+1 คือจำนวน PMP ที่พัฒนาขึ้น รีจิสเตอร์ที่อยู่ถูกเลื่อนขวา 2 บิต เนื่องจาก PMP มีความละเอียด 4 ไบต์ (four-byte granularity) รีจิสเตอร์กำหนดค่าถูกบรรจุอย่างหนาแน่นใน CSRs เพื่อเร่งการสลับบริบท (context switching) ตามที่แสดงรูปที่ 11.12 การกำหนดค่าของ PMP ประกอบด้วยบิต R, W และ X ซึ่งเมื่อตั้งค่าแล้วจะอนุญาตให้โหลด (loads), บันทึก (stores), และดึงคำสั่ง (fetches) ตามลำดับ และฟิลด์โหมด A ซึ่งเมื่อเป็น 0 จะปิดใช้งาน PMP นี้ และเมื่อเป็น 1 จะเปิดใช้งาน การกำหนดค่า PMP ยังรองรับโหมดอื่นๆ และสามารถถูกล็อกได้ คุณสมบัติเหล่านี้มีคำอธิบายในเอกสาร [Waterman and Asanović 2017]



รูปที่ 11.13: รีจิสเตอร์การมอบหมายข้อยกเว้น รีจิสเตอร์การมอบหมายข้อยกเว้น และการขัดจังหวะของโหมดเครื่องจักรและโหมดหัวหน้างาน (`medeleg`, `sedeleg`, `sedeleg`, `mideleg`, `sideleg`) รีจิสเตอร์เหล่านี้ ช่วยให้สามารถมอบหมายการจัดการไปยังตัวจัดการการดักจับ (trap handler) ที่มีสิทธิพิเศษต่ำกว่า โดยตำแหน่งบิตที่เปิดใช้งานจะสอดคล้องกับข้อยกเว้น หรือการขัดจังหวะในรีจิสเตอร์ [m/s]ip

11.5 โหมดหัวหน้างานสำหรับระบบปฏิบัติการสมัยใหม่

กลไก PMP ที่อธิบายในส่วนก่อนหน้านี้เหมาะสำหรับระบบฝังตัว เนื่องจากให้การป้องกันหน่วยความจำด้วยต้นทุนที่ค่อนข้างต่ำ แต่มีข้อจำกัดหลายประการที่จำกัดการใช้งานในระบบประมวลผลทั่วไป (general-purpose computing) เนื่องจาก PMP รองรับเฉพาะจำนวนภูมิภาคหน่วยความจำที่คงที่ (fixed number of memory regions) จึงไม่สามารถปรับขนาดได้สำหรับแอปพลิเคชันที่ซับซ้อน และเนื่องจากขอบเขตเหล่านี้ต้องต่อเนื่องกันในหน่วยความจำกายภาพ ระบบอาจประสบปัญหาการกระจายตัวของหน่วยความจำ (memory fragmentation) นอกจากนี้ PMP ไม่สนับสนุนการแบ่งหน้า (paging) ไปยังหน่วยความจำรอง (secondary storage) ได้อย่างมีประสิทธิภาพ

การกระจายตัวของหน่วยความจำเกิดขึ้นเมื่อหน่วยความจำว่าง แต่ไม่ใช่ส่วนที่ต่อเนื่องกันขนาดใหญ่พอที่จะใช้งานได้

โปรเซสเซอร์ RISC-V ที่ ซับ ซ้อน มาก ขึ้น จัดการ ปัญหา เหล่า นี้ ใน วิธี เดียว กับสถาปัตยกรรม ประมวล ผล ทั่วไป เกือบ ทั้งหมด: โดย ใช้ หน่วย ความ จำ เสมือน แบบ แบ่งหน้า (page-based virtual memory) คุณสมบัตินี้เป็นแกนหลักของ โหมด หัวหน้า งาน (Supervisor mode, S-mode) ซึ่งเป็นโหมดสิทธิพิเศษเสริมที่ออกแบบมาเพื่อสนับสนุนระบบปฏิบัติการสมัยใหม่ เช่น Linux, FreeBSD และ Windows S-mode มีสิทธิพิเศษมากกว่า U-mode แต่มีสิทธิพิเศษน้อยกว่า M-mode เช่นเดียวกับ U-mode ซอฟต์แวร์ใน S-mode ไม่สามารถใช้คำสั่งหรือ CSRs ของ M-mode และอยู่ภายใต้ข้อจำกัดของ PMP ส่วนนี้ครอบคลุมการขัดจังหวะและข้อยกเว้นของ S-mode และส่วนถัดไปจะอธิบายรายละเอียดเกี่ยวกับระบบหน่วยความจำเสมือนของ S-mode

ทำไมไม่มอบหมายการขัดจังหวะให้ S-mode โดยไม่มีเงื่อนไข? เหตุผลหนึ่งคือการจำลองเสมือน หาก M-mode ต้องการจำลองอุปกรณ์ (virtualize a device) สำหรับ S-mode การขัดจังหวะของอุปกรณ์นั้นควรไปที่ M-mode ไม่ใช่ S-mode

โดยค่าเริ่มต้น ข้อยกเว้นทั้งหมด ไม่ว่าจะอยู่ในโหมดสิทธิพิเศษใด จะถ่ายโอนการควบคุมไปยังตัวจัดการข้อยกเว้นของ M-mode (M-mode exception handler) อย่างไรก็ตาม ในระบบ Unix ข้อยกเว้นส่วนใหญ่ควรเรียกระบบปฏิบัติการ ซึ่งทำงานใน S-mode ตัวจัดการข้อยกเว้นของ M-mode สามารถเปลี่ยนเส้นทางข้อยกเว้นไปยัง S-mode ได้ แต่โค้ดเพิ่มเติมนี้จะทำให้การจัดการข้อยกเว้นส่วนใหญ่ช้าลง ดังนั้น RISC-V จึงมีกลไก การมอบหมายข้อยกเว้น (exception delegation) ซึ่งช่วยให้การขัดจังหวะและข้อยกเว้นซิงโครนัส (synchronous exceptions) สามารถมอบหมายไปยัง S-mode ได้โดยเลือกสรรโดยไม่ผ่านซอฟต์แวร์ M-mode เลย

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
Reserved	SEIP	*Res.*	*Res.*	STIP	*Res.*	*Res.*	SSIP	*Res.*
Reserved	SEIE	*Res.*	*Res.*	STIE	*Res.*	*Res.*	SSIE	*Res.*
XLEN-10	1	1	2	1	1	2	1	1

รูปที่ 11.14: รีจิสเตอร์การขัดจังหวะของโหมดหัวหน้างาน รีจิสเตอร์เหล่านี้เป็นรีจิสเตอร์อ่าน/เขียน ขนาด (XLEN) บิต ที่เก็บการขัดจังหวะที่รอดำเนินการ (`sip`) และบิตเปิดใช้งานการขัดจังหวะ (`sie`)

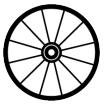
รีจิสเตอร์ควบคุมสถานะ `mideleg` (*Machine Interrupt Delegation*) ควบคุมว่าการขัดจังหวะใดจะถูกมอบหมาย (delegated) ไปยังโหมดหัวหน้างาน (รูปที่ 11.13) เช่นเดียวกับ `mip` และ `mie` แต่ละบิตใน `mideleg` สอดคล้องกับรหัสข้อยกเว้นที่มีหมายเลขเดียวกัน ในรูปที่ 11.3 ตัวอย่างเช่น `mideleg`[5] สอดคล้องกับการขัดจังหวะตัวจับเวลา (timer interrupt) ของ S-mode หากตั้งค่าไว้ การขัดจังหวะตัวจับเวลาใน S-mode จะถ่ายโอนการควบคุมไปยังตัวจัดการข้อยกเว้นของ S-mode แทนตัวจัดการข้อยกเว้นของโหมดเครื่องจักร

S-mode ไม่ สามารถควบคุมการขัดจังหวะตัวจับเวลาหรือการขัดจังหวะซอฟต์แวร์ได้โดยตรง แต่จะใช้คำสั่ง `ecall` เพื่อร้องขอให้ M-mode ตั้งค่าตัว จับ เวลา หรือ ส่งการ ขัดจังหวะ ระหว่างโปรเซสเซอร์ (interprocessor interrupts) ในนามของมัน แนวปฏิบัตินี้เป็นส่วนหนึ่งของ SBI Supervisor Binary Interface

การขัดจังหวะใดๆ ที่ถูกมอบหมายไปยัง S-mode สามารถถูกปิดกั้น (masked) โดยซอฟต์แวร์ใน S-mode ได้ รีจิสเตอร์ `sie` และ `sip` เป็น เป็นรีจิสเตอร์ของ S-mode ซึ่งเป็นส่วนย่อยของ `mie` และ `mip` (รูปที่ 11.14) รีจิสเตอร์เหล่านี้มีโครงสร้างเหมือนกับรีจิสเตอร์ใน M-mode แต่เฉพาะบิตที่สอดคล้องกับการขัดจังหวะที่ถูกมอบหมายใน `mideleg` เท่านั้น ที่สามารถอ่านและเขียนได้ผ่าน `sie` และ `sip` บิตที่สอดคล้องกับการขัดจังหวะ ที่ไม่ถูกมอบหมายจะเป็นศูนย์เสมอ

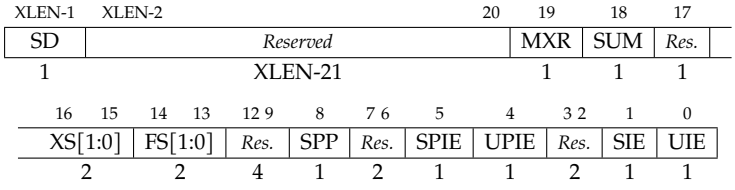
M-mode ยังสามารถมอบหมายข้อยกเว้นซิงโครนัส (synchronous exceptions) ไปยัง S-mode ได้โดยใช้รีจิสเตอร์ `medeleg` (รูปที่ 11.13) กลไกนี้คล้ายกับการมอบหมายการขัดจังหวะ แต่บิตใน `medeleg` สอดคล้องกับรหัสข้อยกเว้นแบบซิงโครนัส ในรูปที่ 11.3 ตัวอย่างเช่น การตั้งค่า `medeleg`[15] จะมอบหมาย store page faults ให้กับ S-mode โดยข้อยกเว้นจะไม่ถ่ายโอนการควบคุมไปยังโหมดที่มีสิทธิพิเศษต่ำกว่า ไม่ว่าการตั้งค่าการมอบหมายจะเป็นอย่างไร ข้อยกเว้นที่เกิดใน M-mode จะถูกจัดการใน M-mode เสมอ ข้อยกเว้นที่เกิดใน S-mode อาจถูกจัดการโดย M-mode หรือ S-mode ขึ้นอยู่กับการกำหนดค่าการมอบหมาย แต่จะไม่ถูกจัดการใน U-mode

S-mode มีรีจิสเตอร์การจัดการข้อยกเว้นหลายตัว ได้แก่ `scause`, `stvec`, `sepc`, `stval`, `sscratch` และ `sstatus` ซึ่งทำหน้าที่เหมือนกับรีจิสเตอร์ใน M-mode ที่อธิบายไว้ในหัวข้อ 11.2 (ดูรูปที่ 11.7 ถึง 11.8). รูปที่ 11.15 แสดงโครงสร้างของรีจิสเตอร์ `sstatus` คำสั่งคืนจากข้อยกเว้นของโหมดหัวหน้างาน `sret` ทำงานเหมือนกับ `mret` แต่จะทำงานกับรีจิสเตอร์การจัดการข้อยกเว้นของ S-mode แทนรีจิสเตอร์ของ M-mode



ความเรียบง่าย

การเกิดข้อยกเว้นใน S-mode มีความคล้ายคลึงกับ M-mode หาก hart (Highway Addressable Remote Transducer) เกิดข้อยกเว้นและถูกมอบหมายให้ S-mode



รูปที่ 11.15: `sstatus` CSR รีจิสเตอร์ `sstatus` เป็นชุดย่อยของ `mstatus` (ดูรูปที่ 11.4) จึงมีโครงสร้างที่คล้ายกัน บิต SIE และ SPIE เก็บสถานะการเปิดใช้งานการขัดจังหวะปัจจุบันและก่อนเกิดข้อยกเว้น (pre-exception interrupt enables) คล้ายกับ MIE และ MPIE ใน `mstatus` XLEN คือ 32 สำหรับ RV32 หรือ 64 สำหรับ RV64 รูปที่ 4.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้ ดูหัวข้อ 4.1 ของเอกสารนั้นสำหรับคำอธิบายของฟิลด์อื่นๆ

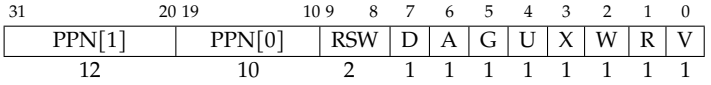
ฮาร์ดแวร์จะทำการเปลี่ยนสถานะแบบอะตอมิก (atomic state transitions) โดยใช้รีจิสเตอร์ของ S-mode แทนรีจิสเตอร์ของ M-mode เมื่อเกิดข้อยกเว้นใน S-mode จะมีการเปลี่ยนแปลงสถานะดังต่อไปนี้:

- PC ของคำสั่งที่ทำให้เกิดข้อยกเว้นจะถูกเก็บไว้ใน `sepc` และ PC จะถูกตั้งค่าเป็น `stvec`
- `scause` ถูกเตั้งค่าเป็นสาเหตุของข้อยกเว้นตามที่เข้ารหัส ในรูปที่ 11.3 และ `stval` ถูกตั้งค่าเป็นที่อยู่ที่ผิดพลาด หรือข้อมูลเฉพาะของข้อยกเว้นอื่นๆ
- การขัดจังหวะถูกปิดใช้งานโดยตั้งค่า SIE=0 ในรีจิสเตอร์ `sstatus` และค่าก่อนหน้าของ SIE ถูกเก็บไว้ใน SPIE
- โหมดสิทธิพิเศษก่อนเกิดข้อยกเว้นถูกเก็บไว้ในฟิลด์ SPP ของ `sstatus` และโหมดสิทธิพิเศษถูกเปลี่ยนเป็น S-mode

11.6 ระบบหน่วยความจำเสมือนใน S-mode

S-mode มีระบบหน่วยความจำเสมือนแบบทั่วไปที่แบ่งหน่วยความจำออกเป็น *หน้า (pages)* ขนาดคงที่เพื่อการแปลที่อยู่และการป้องกันหน่วยความจำ เมื่อเปิดใช้งานการแบ่งหน้า ที่อยู่ส่วนใหญ่ (รวมถึงที่อยู่ที่มีผลของการโหลดและจัดเก็บ และ PC) เป็น *ที่อยู่เสมือน (virtual addresses)* ซึ่งต้องถูกแปลเป็น *ที่อยู่กายภาพ (physical addresses)* การแปลที่อยู่เสมือนเป็นที่อยู่กายภาพทำได้โดยการสำรวจ *ตารางหน้า (page table)* ซึ่งเป็นทรีที่มีฐานสูง (high-radix tree) โหนดใบ (leaf node) ในตารางหน้าจะระบุว่าที่อยู่เสมือนมีการแมปไปยังหน้าหน่วยความจำกายภาพหรือไม่ และหากมีการแมป โหมดสิทธิพิเศษและประเภทการเข้าถึงใดที่ได้รับอนุญาต การเข้าถึงหน้าที่ไม่ได้แมป (unmapped) หรือไม่มีสิทธิ์เพียงพอจะทำให้เกิด ข้อยกเว้นหน้า (page fault exception)

4 KiB **หน้า (pages) ได้รับความนิยมเป็นเวลาห้าทศวรรษ**เริ่มต้นจาก IBM 360 รุ่น 67 คอมพิวเตอร์เครื่องแรกที่มีการแบ่งหน้า Atlas มีหน้าขนาด 3 KiB (ใช้เวิร์ด 6 ไบต์) แล้นที่น่าทึ่งที่หลังจากครึ่งศตวรรษของการเติบโตอย่างก้าวกระโดดในประสิทธิภาพและความจุหน่วยความจำ ขนาดหน้ายังคงแทบไม่เปลี่ยนแปลง



รูปที่ 11.16: รายการตารางหน้า RV32 Sv32 (PTE)

รูปแบบการแบ่งหน้าใน RISC-V ถูกตั้งชื่อเป็น Sv*X* โดยที่ *X* คือขนาดของที่อยู่เสมือนในหน่วยบิต โครงสร้าง RV32 สำหรับ Sv32 รองรับพื้นที่ที่อยู่เสมือน 4 GiB ซึ่งถูกแบ่งออกเป็น 2^{10} *เมกกาเพจ* ขนาด 4 MiB แต่ละเมกะเพจแบ่งย่อยเป็น 2^{10} *หน้าเบื้องต้น (base pages)*—ซึ่งเป็นหน่วยพื้นฐานของการแบ่งหน้า— ขนาด 4 KiB ดังนั้น ตารางหน้าของ Sv32 เป็นทรีสองระดับของที่มีฐาน 2^{10} แต่ละรายการในตารางหน้ามีขนาด 4 ไบต์ ดังนั้นตารางหน้าทั้งหมดมีขนาด 4 KiB การออกแบบนี้ทำให้ขนาดตารางหน้าเท่ากับขนาดหน้าพอดี ซึ่งช่วยให้การจัดสรรหน่วยความจำของระบบปฏิบัติการง่ายขึ้น

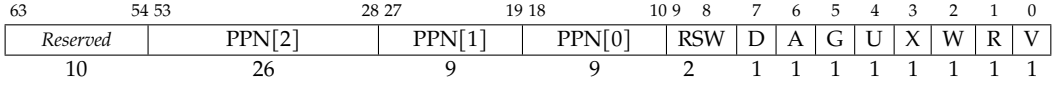
รูปที่ 11.16 แสดงโครงสร้างของรายการตารางหน้า Sv32 (PTE) ซึ่งมีฟิลด์ดังต่อไปนี้ อธิบายจากขวาไปซ้าย:

- บิต V ระบุว่ารายการ PTE นี้ถูกต้องหรือไม่ (V=1) ถ้า V=0 การแปลที่อยู่เสมือนที่ผ่าน PTE นี้จะทำให้เกิดข้อผิดพลาดหน้า (page fault)
- บิต R, W และ X ระบุหน้านั้นมีสิทธิ์อ่าน (read), เขียน (write), และดำเนินการ (execute) ตามลำดับ หากทั้งสามบิตเป็น 0 รายการนี้จะเป็นตัวชี้ไปยังระดับถัดไปของตารางหน้า มิฉะนั้นจะเป็นโหนดใบของทรี
- บิต U ระบุว่าเพจนี้เป็นเพจของผู้ใช้หรือไม่ หาก U=0, U-mode ไม่สามารถเข้าถึงเพจนี้ได้ แต่ S-mode สามารถเข้าถึงได้ หาก U=1, U-mode สามารถเข้าถึงได้ แต่ S-mode ไม่สามารถเข้าถึง
- บิต G ระบุว่าการแมปนี้มีอยู่ในทุกพื้นที่ที่อยู่เสมือน (virtual-address spaces) ซึ่งฮาร์ดแวร์ใช้เพื่อเพิ่มประสิทธิภาพการแปลที่อยู่ โดยทั่วไปใช้สำหรับเพจของระบบปฏิบัติการ
- บิต A ระบุว่าเพจนี้ถูกเข้าถึง (accessed) หรือไม่ตั้งแต่ครั้งสุดท้ายที่บิต A ถูกเคลียร์
- บิต D ระบุว่าเพจนี้ถูกเขียน (dirtied) หรือไม่ตั้งแต่ครั้งสุดท้ายที่บิต D ถูกเคลียร์
- ฟิลด์ RSW สงวนไว้สำหรับการใช้งานของระบบปฏิบัติการ ฮาร์ดแวร์จะละเว้น
- ฟิลด์ PPN เก็บหมายเลขหน้าหน่วยความจำกายภาพ (physical page number) ซึ่งเป็นส่วนหนึ่งของที่อยู่กายภาพ หาก PTE เป็นโหนดใบ PPN จะเป็นส่วนหนึ่งของที่อยู่กายภาพที่แปลได้ มิฉะนั้น PPN จะให้ที่อยู่ของระดับถัดไปของตารางหน้า (รูปที่ 11.16 แบ่ง PPN เป็นสองฟิลด์ย่อยเพื่ออธิบายขั้นตอนการแปลที่อยู่)

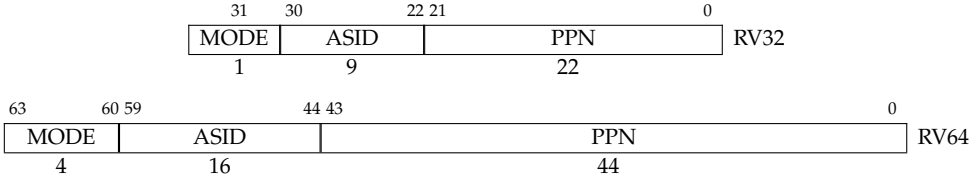
ระบบปฏิบัติการใช้บิต A และ D เพื่อตัดสินใจว่าจะสลับเพจใดไปยังที่เก็บข้อมูลสำรองการล้างบิต A เป็นระยะจะช่วยให้ OS ประมาณว่าหน้าใดมีการใช้งานน้อยที่สุด หน้า D บิตบ่งชี้ว่าการสลับเพจมีราคาแพงกว่า เพราะต้องเขียนกลับไปที่ที่เก็บข้อมูลสำรอง

โครงสร้างการแบ่งหน้าอื่น ๆ สำหรับ RV64เพียงเพิ่มระดับให้กับตารางหน้า Sv48 คล้ายกับ Sv39 แต่พื้นที่ที่อยู่เสมือนใหญ่กว่า 2^9 เท่า และตารางหน้าเพิ่มความลึกอีกหนึ่งระดับ

RV64 รองรับรูปแบบการแบ่งหน้า (paging schemes) หลายแบบ แต่เราจะอธิบายเฉพาะรูปแบบที่ได้รับความนิยมมากที่สุด คือ Sv39 Sv39 ใช้หน้าฐาน (base page) ขนาด



รูปที่ 11.17: รายการตารางหน้า RV64 Sv39 (PTE)



รูปที่ 11.18: รีจิสเตอร์ `satp` รูปที่ 4.11 และ 4.12 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้

4 KiB เช่นเดียวกับ Sv32 รายการตารางหน้า (page-table entries) มีขนาดเพิ่มเป็น 8 ไบต์ เพื่อรองรับที่อยู่กายภาพที่ใหญ่ขึ้น เพื่อรักษาคุณสมบัติที่ตารางหน้าต้องมีขนาดเท่ากับขนาดหน้า ฐานของทรีจึงลดลงเป็น 2^9 ทำให้ทรีมีความลึก 3 ระดับ พื้นที่ที่อยู่ขนาด 512 GiB ของ Sv39 ถูกแบ่งออกเป็น 2^9 *gigapages* โดยแต่ละ gigapage มีขนาด 1 GiB แต่ละ gigapage ถูกแบ่งย่อยเป็น 2^9 megapages ซึ่งใน Sv39 มีขนาดเล็กกว่าใน Sv32 เล็กน้อย คือ 2 MiB และแต่ละ megapages แบ่งย่อยเป็น 2^9 หน้าฐานขนาด 4 KiB

รูปที่ 11.17 แสดงเค้าโครงสร้างของ PTE ใน Sv39 ซึ่งเหมือนกับ Sv32 ยกเว้นฟิลด์ PPN ที่ขยายเป็น 44 บิตเพื่อรองรับที่อยู่กายภาพ 56 บิต หรือ 2^{26} GiB ของพื้นที่ที่อยู่กายภาพ

■ ***รายละเอียดเพิ่มเติม: บิตที่อยู่ที่ไม่ได้ใช้งาน***

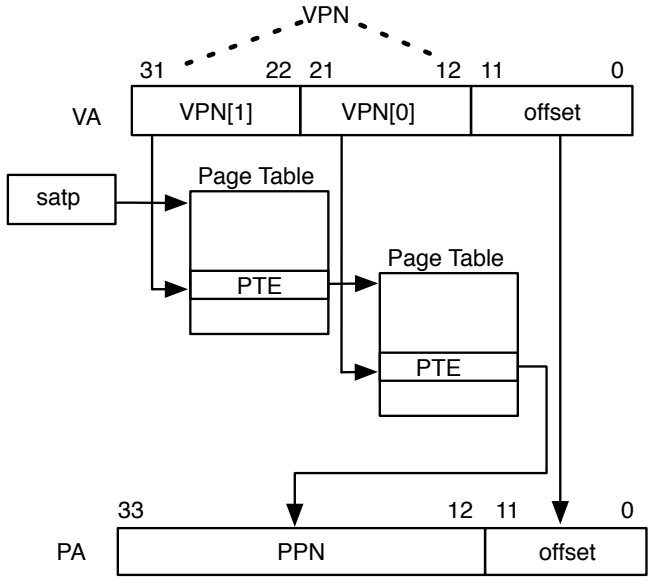
เนื่องจากที่อยู่เสมือนของ Sv39 มีความกว้างน้อยกว่าขนาดของรีจิสเตอร์จำนวนเต็มใน RV64 คุณอาจสงสัยว่าบิตที่เหลืออีก 25 บิตถูกใช้อย่างไร Sv39 กำหนดให้บิตที่ 63–39 เป็นสำเนาของบิตที่ 38 ดังนั้นที่อยู่เสมือนที่ถูกต้องจึงอยู่ในช่วง $\texttt{0000_0000_0000_0000}_{hex}$–$\texttt{0000_003f_ffff_ffff}_{hex}$ และ $\texttt{ffff_ffc0_0000_0000}_{hex}$–$\texttt{ffff_ffff_ffff_ffff}_{hex}$ ช่องว่างระหว่างสองช่วงนี้มีขนาดใหญ่กว่าขนาดของทั้งสองช่วงรวมกันถึง 2^{25} เท่า ซึ่งดูเหมือนจะสูญเสีย 99.999997% ของค่าที่เรจิสเตอร์ 64 บิตสามารถแสดงได้ ทำไมจึงไม่ใช้ประโยชน์จากบิตที่เหลือ 25 บิตนี้ให้ดีกว่านี้? คำตอบคือ เมื่อโปรแกรมเริ่มต้องการพื้นที่ที่อยู่เสมือนมากกว่า 512 GiB สถาปนิกต้องการเพิ่มขนาดพื้นที่ที่อยู่โดยไม่ทำลายความเข้ากันได้ย้อนหลัง (backwards compatibility) หากอนุญาตให้โปรแกรมเก็บข้อมูลเพิ่มเติมในบิตด้านบน 25 บิตจะเป็นไปไม่ได้ที่จะเรียกคืนบิตเหล่านั้นในภายหลังเพื่อใช้สำหรับที่อยู่ที่ใหญ่ขึ้น การอนุญาตให้เก็บข้อมูลในบิตที่อยู่ที่ไม่ได้ใช้งานถือเป็นความผิดพลาดร้ายแรง ซึ่งเกิดขึ้นซ้ำหลายครั้งในประวัติศาสตร์คอมพิวเตอร์

รีจิสเตอร์ควบคุมและสถานะของ S-mode ชื่อ `satp` (*Supervisor Address Translation and Protection*) ใช้ควบคุมระบบการแบ่งหน้า (paging system) ดังที่แสดงในรูปที่ 11.18 รีจิสเตอร์ `satp` มี 3 ฟิลด์ ฟิลด์ MODE เปิดใช้งานการแบ่งหน้าและเลือกความลึกของตารางหน้า ดังที่แสดงในรูปที่ 11.19 ฟิลด์ ASID (*Address Space Identifier*) เป็นฟิลด์เสริมที่สามารถใช้เพื่อลดต้นทุนของการสลับบริบท (context switches) และ ฟิลด์

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

รูปที่ 11.19: การเข้ารหัสของฟิลด์ MODE ในรีจิสเตอร์ควบคุมและสถานะ `satp` Table 4.3 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้



รูปที่ 11.20: ไดอะแกรมของกระบวนการแปลที่อยู่ Sv32

PPN เก็บที่อยู่กายภาพของตารางหน้าหลัก (root page table) หารด้วยขนาดหน้า คือ 4 KiB โดยทั่วไป โดยทั่วไป ซอฟต์แวร์ใน Machine mode (M-mode) จะเขียนค่า 0 ลงใน `satp` ก่อนเข้าสู่ S-mode เป็นครั้งแรกเพื่อปิดใช้งานการแบ่งหน้า จากนั้นซอฟต์แวร์ใน S-mode จะเขียน `satp` อีกครั้งหลังจากตั้งค่าตารางหน้าเสร็จสิ้น

เมื่อการแบ่งหน้าถูกเปิดใช้งานในรีจิสเตอร์ `satp` ที่อยู่เสมือนของโหมด S และ โหมด U ถูกแปลเป็นที่อยู่กายภาพโดยการสำรวจตารางหน้า โดยเริ่มต้นจาก ราก (root) รูปที่ 11.20 แสดงกระบวนการนี้:

1. `satp`.PPN ให้ที่อยู่ฐานของตารางหน้าระดับแรก และ VA[31:22] ให้ดัชนีระดับแรก ดังนั้นโปรเซสเซอร์จะอ่านรายการตารางหน้า (PTE) ที่อยู่ (`satp`.PPN ×4096 + VA[31:22]×4).
2. รายการ PTE มีตำแหน่งที่อยู่ฐานของตารางหน้าระดับที่สอง และ VA[21:12] ให้ดัชนีระดับที่สอง ดังนั้นโปรเซสเซอร์จะอ่านรายการตารางหน้าใบ (leaf PTE) ที่อยู่

(PTE.PPN×4096 + VA[21:12]×4).

3. ฟิลด์ PPN ของรายการตารางหน้าใบและส่วนที่เหลือของหน้า *page offset* (บิต 12 บิตต่ำสุดของที่อยู่เสมือนเดิม) จะรวมกันเป็นผลลัพธ์สุดท้าย: ที่อยู่กายภาพคือ (LeafPTE.PPN×4096 + VA[11:0])

จากนั้นโปรเซสเซอร์จะดำเนินการเข้าถึงหน่วยความจำกายภาพ กระบวนการแปลที่อยู่สำหรับ Sv39 คล้ายกับ Sv32 แต่ใช้ PTE ขนาดใหญ่กว่าและมีระดับการอ้างอิงเพิ่มขึ้นหนึ่งระดับ รูปที่ 11.27 ในท้ายบทนี้ให้คำอธิบายที่สมบูรณ์ของขั้นตอนการสำรวจตารางหน้าโดยระบุเงื่อนไขที่ผิดปกติและกรณีพิเศษของการแปลซูเปอร์เพจ

นั่นคือทั้งหมดของระบบการแบ่งหน้าใน RISC-V ยกเว้นเพียงประเด็นเล็กน้อยหนึ่งประการ หากการดึงคำสั่ง (fetches), การโหลด (loads), และการบันทึก (stores) ทุกครั้งต้องมีการเข้าถึงตารางหน้าหลายครั้ง การแบ่งหน้าจะลดประสิทธิภาพลงอย่างมาก! โปรเซสเซอร์สมัยใหม่ทั้งหมดลดภาระนี้ด้วย แคชการแปลที่อยู่ หรือมักเรียกว่า *TLB* (Translation Lookaside Buffer) เพื่อลดต้นทุนของแคชนี้ โปรเซสเซอร์ส่วนใหญ่ไม่รักษาความสอดคล้องของ TLB กับตารางหน้าโดยอัตโนมัติ—หากระบบปฏิบัติการแก้ไขตารางหน้า แคชจะล้าสมัย (stale) S-mode จึงเพิ่มคำสั่ง `sfence.vma` เพื่อแจ้งให้โปรเซสเซอร์ทราบว่าซอฟต์แวร์อาจมีการแก้ไขตารางหน้า ทำให้โปรเซสเซอร์สามารถล้างแคชการแปล (translation caches) ได้ตามนั้น คำสั่งนี้รับพารามิเตอร์เสริมสองตัวเพื่อจำกัดขอบเขตของการล้างแคช: rs1 ระบุที่อยู่เสมือนที่มีการแปลถูกแก้ไขในตารางหน้า และ rs2 ระบุตัวระบุพื้นที่ที่อยู่ (address-space identifier) ของโปรเซสที่มีการแก้ไขตารางหน้า หากใช้ `x0` สำหรับทั้งสองพารามิเตอร์ แคชการแปลทั้งหมดจะถูกล้าง



■ ***รายละเอียดเพิ่มเติม: ความสอดคล้องของแคชการแปลที่อยู่ในระบบมัลติโปรเซสเซอร์***

คำสั่ง `sfence.vma` มีผลต่อฮาร์ดแวร์การแปลที่อยู่ของ hart ที่รันคำสั่งนั้นเท่านั้น เมื่อ hart หนึ่งเปลี่ยนตารางหน้าที่ hart อื่นใช้ hart แรกต้องใช้การขัดจังหวะระหว่างโปรเซสเซอร์ เพื่อแจ้งให้ hart ที่สองรันคำสั่ง `sfence.vma` ขั้นตอนนี้มักเรียกว่า *TLB shootdown*

11.7 CSR ระบุคุณสมบัติและประสิทธิภาพ

CSR ที่ระบุคุณสมบัติของโปรเซสเซอร์หรือช่วยวัดประสิทธิภาพ CSR มีดังนี้:

- CSR `misa` (Machine ISA) ระบุความกว้างของที่อยู่ของโปรเซสเซอร์ (32, 64 หรือ 128 บิต) และระบุว่าส่วนขยายคำสั่งใดถูกรวมอยู่ (รูปที่ 11.21)
- CSR `mvendorid` (Vendor ID) ระบุรหัสผู้ผลิต JEDEC ของผู้ให้บริการคอร์ (core) (รูปที่ 11.22)
- CSR `marchid` (Machine Architecture ID) ระบุไมโครสถาปัตยกรรมพื้นฐาน การรวม `mvendorid` และ `marchid` ระบุไมโครสถาปัตยกรรมที่พัฒนาขึ้นอย่างไม่ซ้ำกัน (รูปที่ 11.23)

- CSR `mimpid` (Machine Implementation ID) ระบุรุ่นของ *การพัฒนา*ของไมโครสถาปัตยกรรมพื้นฐานใน `marchid` (รูปที่ 11.23)
- CSR `mhartid` (Hart ID) ระบุรหัสจำนวนเต็มของ hart ที่กำลังรันอยู่ (รูปที่ 11.23)

CSR การวัดประสิทธิภาพ:

- CSR `mtime` (Machine Time) เป็นตัวนับแบบเวลาจริง 64 บิต (รูปที่ 11.24)
- CSR `mtimecmp` (Machine Time Compare) ทำให้เกิดการขัดจังหวะเมื่อ `mtime` เท่ากับหรือมากกว่าค่าของมัน (รูปที่ 11.24)
- CSR `mcounteren` และ `scounteren` รีจิสเตอร์ควบคุมการเข้าถึงตัวนับประสิทธิภาพฮาร์ดแวร์ (hardware performance monitor CSRs) สำหรับระดับสิทธิพิเศษถัดไปที่ต่ำกว่า (รูปที่ 11.25)
- CSR ตัวนับประสิทธิภาพฮาร์ดแวร์ 31 ตัว (`mcycle`, `minstret`, `mhpmcounter3`, ..., `mhpmcounter31`) (clock cycles), คำสั่งที่ดำเนินการเสร็จ (instructions retired), และ เหตุการณ์สูงสุด 29 เหตุการณ์ ที่เลือกโดยโปรแกรมเมอร์โดยใช้ `mhpmevent3`,..., `mhpmevent31` (รูปที่ 11.26)

11.8 ข้อสังเกตสรุป

การวิจัยแสดงให้เห็นว่านักออกแบบที่ดีที่สุดออกแบบโครงสร้างที่เร็วกว่า เล็กกว่า เรียบง่ายกว่า ชัดเจนกว่า และกระบวนการออกแบบที่ง่ายกว่า ความแตกต่างระหว่างนักออกแบบชั้นยอดและนักออกแบบทั่วไปอาจมากถึงหนึ่งลำดับความสำคัญ

—Fred Brooks, Jr., 1986. ผู้ได้รับรางวัล Turing Award และเป็นสถาปนิกของคอมพิวเตอร์ตระกูล IBM System/360 ซึ่งแสดงให้เห็นถึงความสำคัญของการแยกสถาปัตยกรรมออกจากการพัฒนา รุ่นต่อมาของสถาปัตยกรรมปี 1964 นั้นยังคงจำหน่ายอยู่ในปัจจุบัน



ความเรียบง่าย

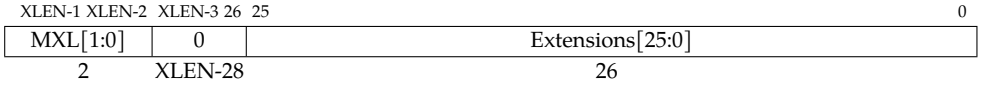


การเขียนโปรแกรม

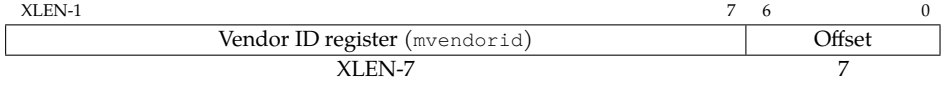
ความเป็นโมดูลของสถาปัตยกรรมที่มีสิทธิพิเศษใน RISC-V ตอบสนองความต้องการของระบบหลากหลายประเภท โหมดเครื่องจักร ที่เรียบง่ายสนับสนุนแอปพลิเคชันฝังตัวแบบ bare-metal ด้วยต้นทุนต่ำ โหมดผู้ใช้ (User mode) และการป้องกันหน่วยความจำกายภาพ (Physical Memory Protection) ร่วมกันช่วยให้สามารถทำงานหลายงาน (multitasking) ในระบบฝังตัวที่ซับซ้อนมากขึ้น และโหมดหัวหน้างานและหน่วยความจำเสมือนแบบหน้า ให้ความยืดหยุ่นที่จำเป็นในการโฮสต์ระบบปฏิบัติการสมัยใหม่

11.9 เรียนรู้เพิ่มเติม

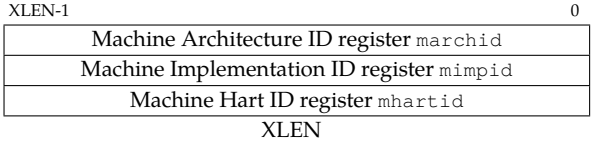
A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017. URL `https://riscv.org/specifications/privileged-isa/`.



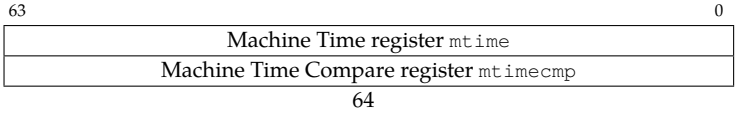
รูปที่ 11.21: CSR `misa` รายงาน ISA ที่รองรับ ฟิลด์ MXL (Machine XLEN) เข้ารหัสความกว้างของ ISA จำนวนเต็มพื้นฐาน: 1 คือ 32 บิต ,2 คือ 64 บิตและ 3 คือ 128 บิต ฟิลด์ Extensions เข้ารหัสการมีอยู่ของส่วนขยายมาตรฐาน โดยใช้บิตเดียวต่อตัวอักษรของตัวอักษร (บิต 0 ระบุส่วนขยาย “A” บิต 1 ระบุส่วนขยาย “B” จนถึงบิต 25 ซึ่งระบุส่วนขยาย “Z”)



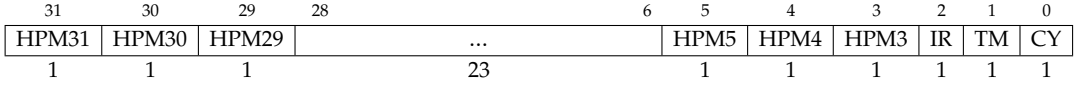
รูปที่ 11.22: CSR `mvendorid` ให้รหัสผู้ผลิต JEDEC ของคอร์



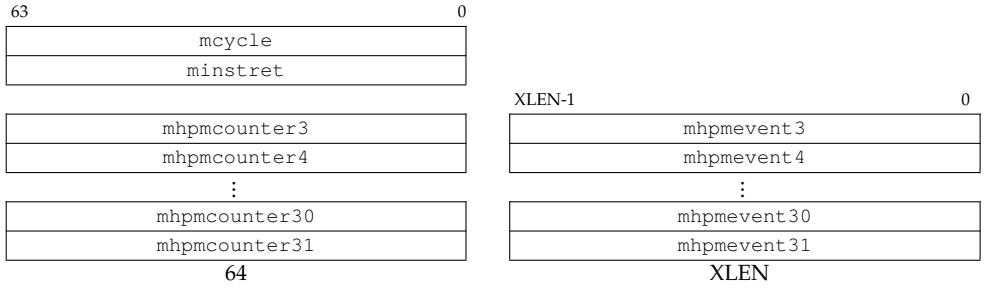
รูปที่ 11.23: CSR ระบุเครื่องจักร (`marchid`, `mimpid`, `mhartid`) ระบุไมโครสถาปัตยกรรมและการพัฒนาของโปรเซสเซอร์ รวมถึงรหัสของ hart ที่กำลังรันอยู่



รูปที่ 11.24: CSR เวลาเครื่องจักร (`mtime` and `mtimecmp`) วัดเวลาและทำให้เกิดการขัดจังหวะเมื่อ `mtime` $\geq$ `mtimecmp`



รูปที่ 11.25: รีจิสเตอร์ควบคุมตัวนับ `mcounteren` และ `scounteren` ควบคุมการเข้าถึงตัวนับประสิทธิภาพฮาร์ดแวร์สำหรับโหมดที่มีสิทธิพิเศษต่ำกว่าถัดไป



รูปที่ 11.26: CSR ตัวนับประสิทธิภาพฮาร์ดแวร์ (`mcycle`, `minstret`, `mhpmcounter3`, ..., `mhpmcounter31`) และเหตุการณ์ที่นับ `mhpmevent3`, ..., `mhpmevent31` สำหรับ RV32 การอ่านค่า CSR ของ `mcycle`, `minstret` และ `mhpmcounter`n จะคืนบิตต่ำ 32 บิต, ส่วนการอ่าน รีจิสเตอร์ `mcycleh`, `minstreth` และ `mhpmcounter`n`h` จะคืนบิต 63–32 ของตัวนับที่เกี่ยวข้อง

1. กำหนดให้ a เป็น `satp`.$ppn \times$ PAGESIZE, และ ให้ $i =$ LEVELS $- 1$
2. กำหนดให้ pte เป็นค่าของ PTE ณ ที่อยู่ $a + va.vpn[i] \times$ PTESIZE
3. ถ้า $pte.v = 0$, หรือ ถ้า $pte.r = 0$ และ $pte.w = 1$, หยุดและเพิ่มข้อยกเว้น page-fault
4. ถ้าไม่ใช่, PTE ไม่ถูกต้อง. ถ้า $pte.r = 1$ หรือ $pte.x = 1$, ไปที่ขั้นตอนที่ 5 มิฉะนั้น PTE นี้จะเป็นตัวชี้ไปยังระดับถัดไปของตารางเพจ กำหนดให้ $i = i - 1$. ถ้า $i < 0$, หยุดและเพิ่มข้อยกเว้น page-fault มิฉะนั้น กำหนดให้ $a = pte.ppn \times$ PAGESIZE ไปที่ขั้นตอนที่ 2
5. พบ leaf PTE ตรวจสอบว่าการเข้าถึงหน่วยความจำที่ร้องขอได้รับอนุญาตโดย $pte.r$, $pte.w$, $pte.x$, และ $pte.u$ บิต, กำหนดโหมดสิทธิพิเศษปัจจุบันและค่าของฟิลด์ SUM และ MXR ของรีจิสเตอร์ `mstatus` ถ้าไม่เช่นนั้น ให้หยุดและเพิ่มข้อยกเว้น page-fault
6. ถ้า $i > 0$ และ $pa.ppn[i - 1 : 0] \neq 0$, นี่คือเพจขนาดใหญ่ที่ไม่ตรงแนว หยุดและเพิ่มข้อยกเว้น page-fault
7. ถ้า $pte.a = 0$, หรือ ถ้า การเข้าถึงหน่วยความจำเป็นที่จัดเก็บ และ $pte.d = 0$, แล้วก็:
   - เพิ่มข้อยกเว้น page-fault หรือ:
   - ตั้งค่าให้ $pte.a$ เป็น 1 และ, หากการเข้าถึงหน่วยความจำเป็นที่เก็บ ให้ตั้งค่าด้วย $pte.d$ เป็น 1
8. การแปลประสบความสำเร็จ ที่อยู่ทางกายภาพที่แปลแล้วจะได้รับดังนี้:
   - $pa.pgoff = va.pgoff$
   - ถ้า $i > 0$, นี่คือการแปลเพจขนาดใหญ่และ $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$

รูปที่ 11.27: ขั้นตอนการแปลที่อยู่เสมือนเป็นที่อยู่กายภาพ va เป็นที่อยู่เสมือนที่ป้อนเข้า และ pa เป็นที่อยู่กายภาพที่ส่งออก ค่าคงที่ PAGESIZE คือ 2^{12} สำหรับ Sv32, LEVELS=2 และ PTESIZE=4 ส่วน Sv39, LEVELS=3 และ PTESIZE=8 หัวข้อที่ 4.3.2 ของ [Waterman and Asanović 2017] เป็นที่มาสำหรับรูปนี้

12 ส่วนขยายเพิ่มเติม RISC-V ในอนาคต

อลัน เพอร์ลิส (1922–1990) เป็น ผู้รับ รางวัล Turing Award คน แรก (1966) ซึ่งมอบให้สำหรับ อิทธิพลของเขาต่อภาษาการ เขียนโปรแกรมขั้นสูง และ คอมไพเลอร์ในปี พ.ศ. 2501 เขาได้ เข้า ร่วม ออกแบบ ALGOL ที่มีอิทธิพลต่อภาษา การเขียนโปรแกรมแบบคำ สั่งเกือบทุกภาษา รวมถึง C และ Java



คนโง่จะละเลยความซับซ้อน นักปฏิบัติยอมทนกับมัน บางคนหลีกเลี่ยงมันได้ แต่อัจฉริยะขจัดมันออกไป
—อลัน เพอร์ลิส, 1982

องค์กร RISC-V จะพัฒนาส่วนขยายเสริมเพิ่มเติมอีกหลายตัว

12.1 ส่วนขยายมาตรฐาน “B” สำหรับการจัดการบิต

ในบทที่ 9 เราได้ครอบคลุมชุดย่อยมาตรฐานสี่ชุดของ RV32B แต่ยังมีอีกห้าชุดที่กำลังพิจารณา ตัวอย่างคำสั่งรวมถึง การเก็บดับเบิลเวิร์ดในรีจิสเตอร์เดียว การสับเปลี่ยน การเรียงลำดับ การวางฟิลด์บิต การบีบอัด และการย้ายตามเงื่อนไข

12.2 ส่วนขยายมาตรฐาน “J” สำหรับภาษาที่แปลแบบไดนามิก

ภาษายอดนิยมหลายภาษาจำนวนมากมักถูกนำไปใช้งานผ่านการแปลแบบไดนามิก รวมถึง Java และ Javascript ภาษาเหล่านี้สามารถได้รับประโยชน์จาก ISA ที่สนับสนุนการตรวจสอบแบบไดนามิกและการเก็บรวบรวม (J ย่อมาจาก *Just-In-Time* คอมไพเลอร์))

12.3 ส่วนขยายมาตรฐาน “P” สำหรับคำสั่ง Packed-SIMD

ส่วนขยาย P แบ่งย่อยรีจิสเตอร์สถาปัตยกรรมที่มีอยู่ออก เพื่อให้การคำนวณแบบขนานข้อมูลบนประเภทข้อมูลที่เล็กลง การออกแบบ Packed-SIMD เป็นจุดออกแบบที่สมเหตุสมผลเมื่อใช้ทรัพยากร datapath กว้างที่มีอยู่แล้ว อย่างไรก็ตาม หากต้องการมูลเพิ่มเติมจำนวนมากเกี่ยวกับการดำเนินการแบบขนานข้อมูล บทที่ 8 แสดงให้เห็นว่าการออกแบบสำหรับสถาปัตยกรรมเวกเตอร์เป็นตัวเลือกที่ดีกว่า และสถาปนิกควรใช้ส่วนขยาย RVV

12.4 ส่วนขยายมาตรฐาน “Zk” สำหรับการเข้ารหัสลับแบบสเกลาร์

ส่วนขยายนี้ให้ฟังก์ชันพื้นฐานที่เป็นประโยชน์สำหรับชุดอัลกอริทึมของ NIST, ความหน่วงเวลาการดำเนินการคงที่, และการสร้างบิตสุ่มแบบกำหนดได้

12.5 ข้อสังเกตสรุป

ทำให้ง่าย ไม่ทำให้มันซับซ้อน

—เฮนรี เดวิด ธอโร นักเขียนชื่อดังแห่งศตวรรษที่ 19, 1854



ที่ว่างสำหรับส่วนขยาย

การมีแนวทางคณะกรรมการแบบเปิดและเหมือนมาตรฐานในการขยาย RISC-V คาดว่าจะหมายความว่าการตอบกลับและการถกเถียงจะเกิดขึ้น *ก่อน*ที่คำสั่งจะถูกกำหนดเป็นที่สิ้นสุด แทนที่จะเป็นหลังจากนั้น เมื่อมันสายเกินไปที่จะเปลี่ยนแปลง ในกรณีที่เหมาะสม สมาชิกบางคนจะนำข้อเสนอไปใช้ก่อนที่มันจะได้รับการรับรอง ซึ่ง FPGA ทำให้ง่ายขึ้นมาก การเสนอส่วนขยายคำสั่งผ่านคณะกรรมการ RISC-V International จะเป็นงานที่ต้องใช้ความพยายามพอสมควร ซึ่งจะทำให้อัตราการเปลี่ยนแปลงช้าลง ไม่เหมือนกับที่เกิดขึ้นกับ x86-32 (ดูรูปที่ 1.2 หน้าที่ 4 บทที่ 1) อย่าลืมว่าทุกอย่างในบทนี้จะเป็นตัวเลือก ไม่ว่าจะมีการนำส่วนขยายมาใช้กี่ตัว



สง่างาม

เราหวังว่า RISC-V จะสามารถพัฒนาไปพร้อมกับความต้องการทางเทคโนโลยี ในขณะที่รักษาชื่อเสียงในฐานะ ISA ที่เรียบง่ายและมีประสิทธิภาพ หากประสบความสำเร็จ RISC-V จะเป็นการเปลี่ยนแปลงครั้งสำคัญจาก ISA แบบเพิ่มหน่วยของอดีต

A รายการคำสั่ง RISC-V

โคโค ชาแนล (1883-1971) ผู้ก่อตั้งแบรนด์แฟชั่นชาแนล การแสวงหาความเรียบง่ายที่มีราคาแพงของเธอ ได้กำหนดรูปแบบแฟชั่นแห่งศตวรรษที่ 20



ความเรียบง่ายคือหัวใจสำคัญของความสง่างามที่แท้จริง

—โคโค ชาแนล, 1923

ภาคผนวกนี้แสดงรายการคำสั่งทั้งหมดสำหรับ RV32/64I รวมถึงส่วนขยายทั้งหมดที่ครอบคลุมในหนังสือเล่มนี้ ยกเว้น RVV (RVM, RVA, RVF, RVD และ RVC) และคำสั่งเทียมทั้งหมด แต่ละรายการประกอบด้วยชื่อคำสั่ง, ตัวดำเนินการ, คำจำกัดความในระดับการถ่ายโอนรีจิสเตอร์, ประเภทของรูปแบบคำสั่ง คำอธิบายคำสั่งภาษาไทย คำสั่งบีบอัด(ถ้ามี) และไดอะแกรมเค้าโครงจริงพร้อมรหัสคำสั่ง (opcodes) ซึ่งเป็นสิ่งที่จำเป็นในการทำความเข้าใจคำสั่งทั้งหมด ในสรุปขนาดกะทัดรัดเหล่านี้ อย่างไรก็ตาม หากคุณต้องการรายละเอียดเพิ่มเติม สามารถอ้างอิงถึงข้อกำหนดอย่างเป็นทางการของ RISC-V [Waterman and Asanović 2017]

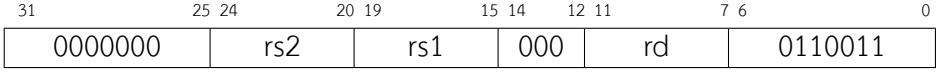
เพื่อช่วยให้ผู้อ่านพบคำสั่งที่ต้องการในภาคผนวกนี้ ส่วนหัวของหน้าซ้าย (คู่) มีคำสั่งแรกจากด้านบนของหน้านั้น และส่วนหัวของหน้าทางหน้าขวา (คี่) จะมีคำสั่งสุดท้ายจากด้านล่างสุดของหน้าหนังสือ รูปแบบคล้ายกับส่วนหัวของพจนานุกรม ซึ่งช่วยให้คุณค้นหาหน้าที่คำสั่งที่คุณต้องการหา ตัวอย่างเช่น ส่วนหัวของหน้าคู่ถัดไปแสดง **AMOADD.D** คำสั่งแรกบนหน้า และส่วนหัวของหน้าคี่ต่อไปแสดง **AMOMAXU.W** ซึ่งเป็นคำสั่งสุดท้ายในหน้านั้น นี่คือสองหน้าที่คุณจะพบคำสั่ง 8 คำสั่งเหล่านี้ `amoadd.d`, `amoadd.w`, `amoand.d`, `amoand.w`, `amomax.d`, `amomax.w`, `amomaxu.d` และ `amomaxu.w`

add rd, rs1, rs2 `x[rd] = x[rs1] + x[rs2]`

บวก (Add). รูปแบบ-R, RV32I และ RV64I.

บวก รีจิสเตอร์ x[*rs2*] กับรีจิสเตอร์ x[*rs1*] และเขียนผลลัพท์ลงใน x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.add** rd, rs2; **c.mv** rd, rs2

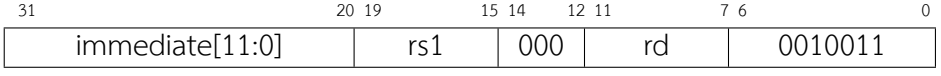


addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

บวกค่าคงที่ทันที (Add Immediate). รูปแบบ-I, RV32I และ RV64I.

บวก *ค่าคงที่*ที่ขยายเครื่องหมาย กับรีจิสเตอร์ x[*rs1*] และเขียนผลลัพท์ลงใน x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

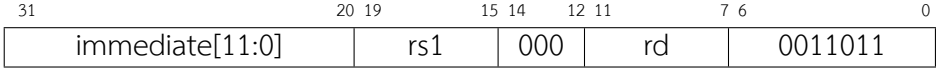


addiw rd, rs1, immediate `x[rd] = sext((x[rs1] + sext(immediate))[31:0])`

บวกเวิร์ดทันที (Add Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

บวก *ค่าคงที่*ที่ขยายเครื่องหมายเข้ากับ x[*rs1*] ตัดทอนผลลัพธ์ให้เหลือ 32 บิต และเขียนผลลัพท์ที่ขยายเครื่องหมายลงใน x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.addiw** rd, imm

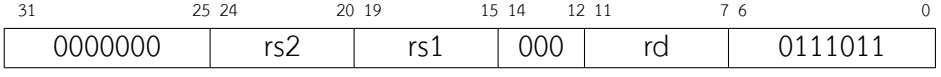


addw rd, rs1, rs2 `x[rd] = sext((x[rs1] + x[rs2])[31:0])`

บวกเวิร์ด (Add Word). รูปแบบ-R, RV64I เท่านั้น.

บวก รีจิสเตอร์ x[*rs2*] เข้ากับรีจิสเตอร์ x[*rs1*] ตัดทอนผลลัพธ์เป็น 32 บิต และเขียนผลลัพท์ที่ขยายเครื่องหมายลงใน x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

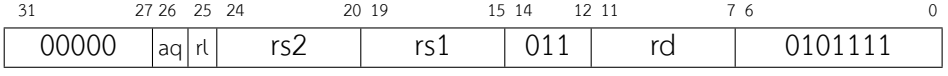
คำสั่งบีบอัด: **c.addw** rd, rs2



amoadd.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] + x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: บวกดับเบิลเวิร์ด (Add Doubleword). รูปแบบ-R, RV64A เท่านั้น.

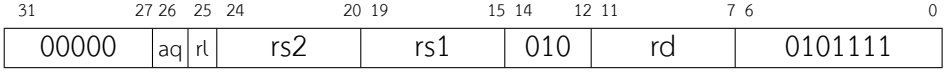
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ด เป็น *t* + x[*rs2*] ตั้งค่า x[*rd*] เป็น *t*



amoadd.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] + x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: บวกเวิร์ด (Add Word). รูปแบบ-R, RV32A และ RV64A.

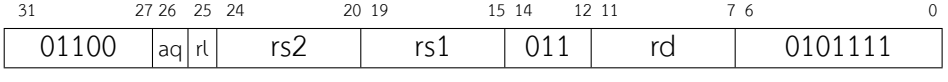
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ด เป็น *t* + x[*rs2*] ตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amoand.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] & x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: AND ดับเบิลเวิร์ด (AND Doubleword). รูปแบบ-R, RV64A เท่านั้น.

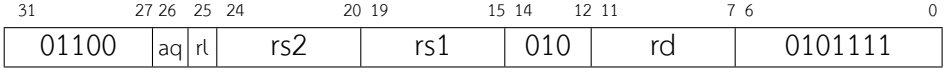
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็นค่าการดำเนินการบิต AND ระหว่าง *t* และ x[*rs2*] ตั้งค่า x[*rd*] เป็น *t*



amoand.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] & x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: AND เวิร์ด (AND word). รูปแบบ-R, RV32A และ RV64A.

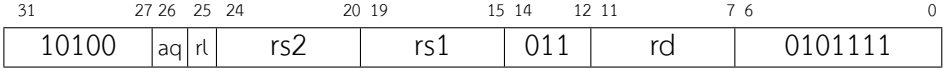
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าการดำเนินการบิต AND ระหว่าง *t* และ x[*rs2*] ตั้งค่า x[*rd*] เป็น *t*



amomax.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MAX x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดดับเบิลเวิร์ด (Maximum Doubleword). รูปแบบ-R, RV64A เท่านั้น.

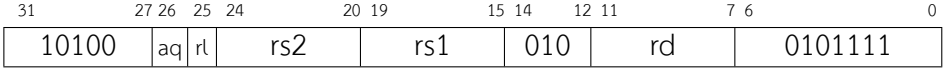
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็นค่าสูงสุดของ *t* และ x[*rs2*] โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองตั้งค่า x[*rd*] เป็น *t*



amomax.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MAX x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: เวิร์ดสูงสุด (Maximum Word). รูปแบบ-R, RV32A และ RV64A.

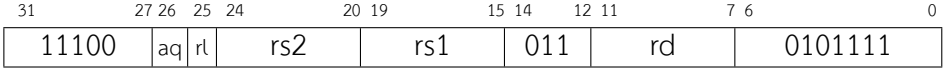
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* คือค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าสูงสุดระหว่าง *t* และ x[*rs2*] โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amomaxu.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดดับเบิลเวิร์ด, ไม่มีเครื่องหมาย (Maximum Doubleword, Unsigned). รูปแบบ-R, RV64A เท่านั้น.

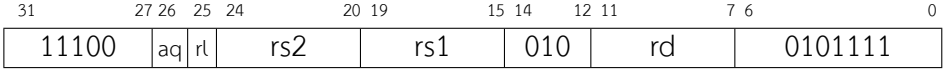
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็นค่าสูงสุดระหว่าง *t* และ x[*rs2*] โดยการเปรียบเทียบแบบไม่มีเครื่องหมาย ตั้งค่า x[*rd*] เป็น *t*



amomaxu.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดเวิร์ด, ไม่มีเครื่องหมาย (Maximum Word, Unsigned). รูปแบบ-R, RV32A และ RV64A.

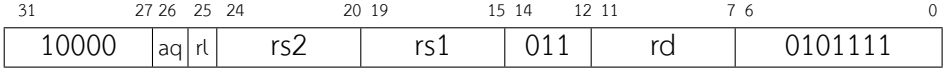
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าสูงสุดระหว่าง *t* และ x[*rs2*] โดยการเปรียบเทียบแบบไม่มีเครื่องหมาย ตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amomin.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MIN x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดดับเบิลเวิร์ด (Minimum Doubleword). รูปแบบ-R, RV64A เท่านั้น.

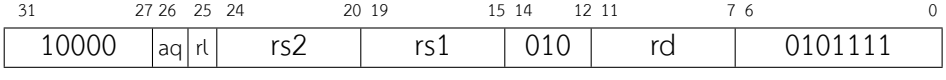
ทำงานแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ด เป็นค่าต่ำสุดระหว่าง *t* และ x[*rs2*] โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองตั้งค่า x[*rd*] เป็น *t*



amomin.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MIN x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดเวิร์ด (Minimum Word). รูปแบบ-R, RV32A และ RV64A.

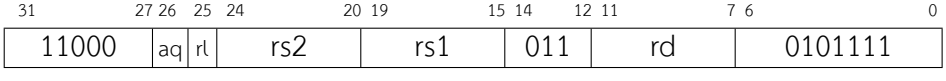
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าต่ำสุดระหว่าง *t* และ x[*rs2*] โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองกำหนดให้ x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amominu.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MINU x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดดับเบิลเวิร์ด ,ไม่มีเครื่องหมาย (Minimum Doubleword, Unsigned). รูปแบบ-R, RV64A เท่านั้น.

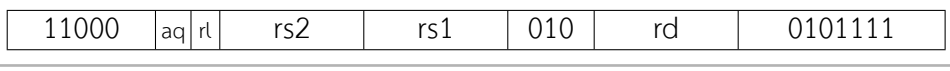
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ด เป็นค่าต่ำสุดระหว่าง *t* และ x[*rs2*] โดยการเปรียบเทียบแบบไม่มีเครื่องหมาย ตั้งค่า x[*rd*] เป็น *t*



amominu.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MINU x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดเวิร์ด, ไม่มีเครื่องหมาย (Minimum Word, Unsigned). รูปแบบ-R, RV32A และ RV64A.

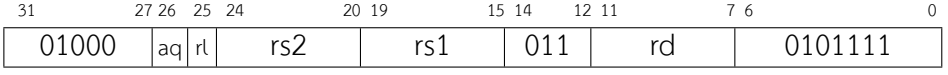
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าต่ำสุดระหว่าง *t* และ x[*rs2*] โดยการเปรียบเทียบแบบไม่มีเครื่องหมาย กำหนดให้ x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amoor.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] | x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: OR ดับเบิลเวิร์ด (OR Doubleword). รูปแบบ-R, RV64A เท่านั้น.

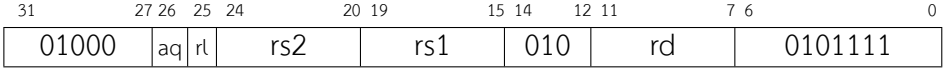
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็นค่าการดำเนินการบิต OR ระหว่าง *t* และ x[*rs2*] ตั้งค่า x[*rd*] เป็น *t*



amoor.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] | x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: OR เวิร์ด (OR Word). รูปแบบ-R, RV32A และ RV64A.

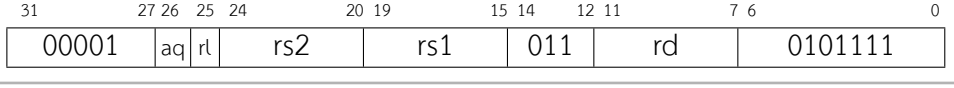
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าการดำเนินการบิต OR ระหว่าง *t* และ x[*rs2*] ตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amoswap.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: สลับดับเบิลเวิร์ด (Swap Doubleword). รูปแบบ-R, RV64A เท่านั้น.

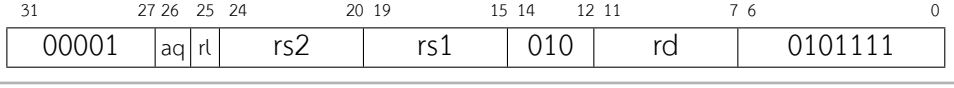
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็น x[*rs2*] ตั้งค่า x[*rd*] เป็น *t*



amoswap.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: สลับเวิร์ด (Swap Word). รูปแบบ-R, RV32A และ RV64A.

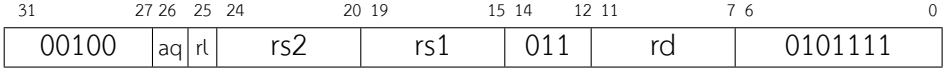
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็น x[*rs2*] ตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*



amoxor.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] ^ x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: XOR ดับเบิลเวิร์ด (XOR Doubleword). รูปแบบ-R, RV64A เท่านั้น.

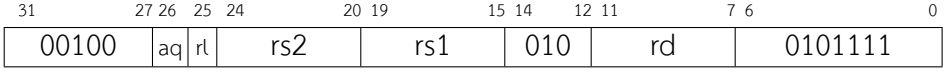
ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 64 บิต (doubleword) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำดับเบิลเวิร์ดเป็นค่าการดำเนินการบิต XOR ระหว่าง *t* และ x[*rs2*] กำหนดให้ x[*rd*] เป็น *t*



amoxor.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] ^ x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: XOR เวิร์ด (XOR Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิก: กำหนดให้ *t* เป็นค่า 32 บิต (word) ของหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] จากนั้น ตั้งค่าหน่วยความจำเวิร์ดเป็นค่าการดำเนินการบิต XOR ระหว่าง *t* และ x[*rs2*] ตั้งค่า x[*rd*] เป็นส่วนขยายเครื่องหมายของ *t*

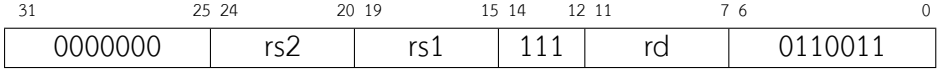


and rd, rs1, rs2 `x[rd] = x[rs1] & x[rs2]`

AND. รูปแบบ-R, RV32I และ RV64I.

คำนวณค่าการดำเนินการบิต AND ระหว่าง รีจิสเตอร์ x[*rs1*] และ รีจิสเตอร์ x[*rs2*] และเขียนผลลัพท์ไปยังรีจิสเตอร์ x[*rd*].

คำสั่งปีบอัด: **c.and** rd, rs2

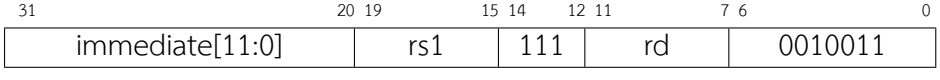


andi rd, rs1, immediate `x[rd] = x[rs1] & sext(immediate)`

AND ค่าคงที่ทันที (AND Immediate). รูปแบบ-I, RV32I และ RV64I.

คำนวณค่าการดำเนินการบิต AND ระหว่าง *ค่าคงที่*ที่ขยายเครื่องหมาย และ รีจิสเตอร์ x[*rs1*] และเขียนผลลัพท์ไปยังรีจิสเตอร์ x[*rd*].

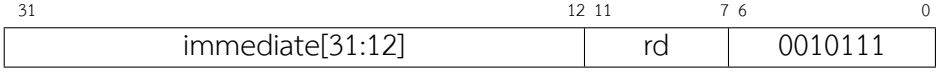
คำสั่งปีบอัด: **c.andi** rd, imm



auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

บวกบิตบนทันทีกับ PC (Add Upper Immediate to PC). รูปแบบ-U, RV32I และ RV64I.

บวก *ค่าคงที่* 20 บิตที่ขยายเครื่องหมาย และเลื่อนซ้าย 12 บิต เข้ากับ (*pc*) และเขียนผลลัพธ์ลงใน x[*rd*]



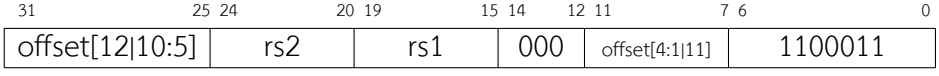
---

beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

กระโดดถ้าเท่ากัน (Branch if Equal). รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าใน รีจิสเตอร์ x[*rs1*] เท่ากับค่าใน รีจิสเตอร์ x[*rs2*] ตั้งค่า *pc* เป็นค่า *pc* ปัจจุบันบวกกับ *offset* ที่ขยายเครื่องหมาย

คำสั่งบีบอัด: **c.beqz** rs1, offset



---

beqz rs1, offset `if (rs1 == 0) pc += sext(offset)`

กระโดดถ้าเท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

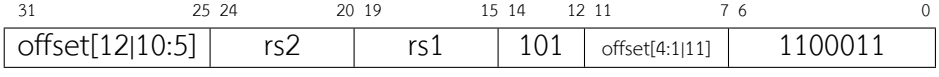
beq rs1, x0, offset

---

bge rs1, rs2, offset `if (rs1` $\geq_s$ `rs2) pc += sext(offset)`

กระโดดถ้ามากกว่าหรือเท่ากับ. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าใน รีจิสเตอร์ x[*rs1*] มีค่ามากกว่าหรือเท่ากับค่าใน รีจิสเตอร์ x[*rs2*] ให้ถือว่าเป็นจำนวนแบบส่วนเติมเต็มสองตั้งค่า *pc* เป็น *pc* ปัจจุบันบวกกับ *offset* ที่ขยายครื่องหมาย

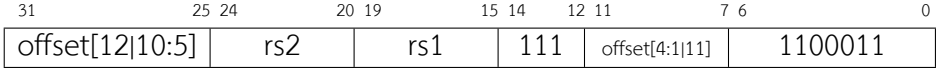


---

bgeu rs1, rs2, offset `if (rs1` $\geq_u$ `rs2) pc += sext(offset)`

กระโดดถ้ามากกว่าหรือเท่ากับ, ไม่มีเครื่องหมาย. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าใน รีจิสเตอร์ x[*rs1*] มีค่ามากกว่าหรือเท่ากับค่าใน x[*rs2*] ให้ถือว่าเป็นจำนวนแบบไม่มีเครื่องหมาย ตั้งค่า *pc* เป็น *pc* ปัจจุบันบวกกับ *offset* ที่ขยายเครื่องหมาย



---

bgez rs1, offset `if (rs1 ≥s 0) pc += sext(offset)`

กระโดดถ้ามากกว่าหรือเท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

bge rs1, x0, offset

---

bgt rs1, rs2, offset `if (rs1 >s rs2) pc += sext(offset)`

กระโดดถ้ามากกว่า. คำสั่งเทียม, RV32I และ RV64I.

blt rs2, rs1, offset

---

bgtu rs1, rs2, offset `if (rs1 >u rs2) pc += sext(offset)`

กระโดดถ้ามากกว่า, ไม่มีเครื่องหมาย. คำสั่งเทียม, RV32I และ RV64I.

bltu rs2, rs1, offset

---

bgtz rs2, offset `if (rs2 >s 0) pc += sext(offset)`

กระโดดถ้ามากกว่าศูนย์. คำสั่งเทียม, RV32I และ RV64I.

blt x0, rs2, offset

---

ble rs1, rs2, offset `if (rs1 ≤s rs2) pc += sext(offset)`

กระโดดถ้าน้อยกว่าหรือเท่ากับ. คำสั่งเทียม, RV32I และ RV64I.

bge rs2, rs1, offset

---

bleu rs1, rs2, offset `if (rs1 ≤u rs2) pc += sext(offset)`

กระโดดถ้าน้อยว่าหรือเท่ากับ, ไม่มีเครื่องหมาย. คำสั่งเทียม, RV32I และ RV64I.

bgeu rs2, rs1, offset

---

blez rs2, offset `if (rs2 ≤s 0) pc += sext(offset)`

กระโดดถ้าน้อยว่าหรือเท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

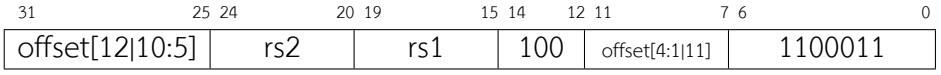
bge x0, rs2, offset

---

blt rs1, rs2, offset `if (rs1 <s rs2) pc += sext(offset)`

กระโดดถ้าน้อยกว่า. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าใน รีจิสเตอร์ x[*rs1*] น้อยกว่าค่าใน รีจิสเตอร์ x[*rs2*] ให้ถือว่าค่าเป็นจำนวนแบบส่วนเติมเต็มสองให้ค่า *pc* เป็น *pc* ปัจจุบันบวกกับส่วนขยายเครื่องหมาย *offset*

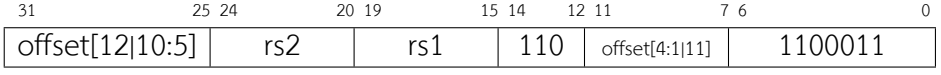


---

bltu rs1, rs2, offset `if (rs1 <u rs2) pc += sext(offset)`

กระโดดถ้าน้อยกว่า, ไม่มีเครื่องหมาย. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าใน รีจิสเตอร์ x[*rs1*] น้อยกว่าค่าใน รีจิสเตอร์ x[*rs2*] ให้ถือว่าค่าเป็นตัวเลขที่ไม่มีเครื่องหมาย ให้ค่า *pc* เป็น *pc* ปัจจุบันบวกกับส่วนขยายเครื่องหมาย *offset*



---

bltz rs1, offset `if (rs1 <s 0) pc += sext(offset)`

กระโดดถ้าน้อยกว่าศูนย์. คำสั่งเทียม, RV32I และ RV64I.

blt rs1, x0, offset

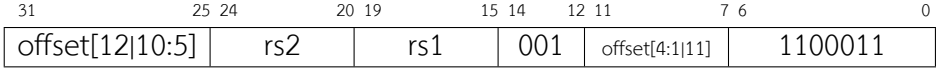
---

bne rs1, rs2, offset `if (rs1 ≠ rs2) pc += sext(offset)`

กระโดดถ้าไม่เท่ากับ. รูปแบบ-B, RV32I และ RV64I.

ถ้า รีจิสเตอร์ x[*rs1*] ไม่เท่ากับ รีจิสเตอร์ x[*rs2*] ตั้งค่า *pc* เป็น *pc* ปัจจุบันบวกกับ ส่วนขยายเครื่องหมาย *offset*

คำสั่งปีบอัด: **c.bnez** rs1, offset



---

bnez rs1, offset `if (rs1 ≠ 0) pc += sext(offset)`

กระโดดถ้าไม่เท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

bne rs1, x0, offset

---

c.add rd, rs2 `x[rd] = x[rd] + x[rs2]`

บวก *(Add).* RV32IC และ RV64IC.

ขยายเป็น **add** rd, rd, rs2. ไม่ถูกต้องเมื่อ rd=x0 หรือ rs2=x0



---

c.addi rd, imm `x[rd] = x[rd] + sext(imm)`

บวกค่าคงที่ *(Add Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **addi** rd, rd, imm

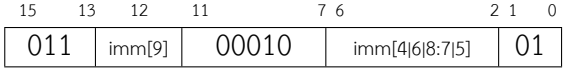


---

c.addi16sp imm `x[2] = x[2] + sext(imm)`

บวกค่าคงที่ ปรับขนาดเป็น 16 ไปยังตัวชี้สแต็ก RV32IC และ RV64IC.

ขยายเป็น **addi** x2, x2, imm. ไม่ถูกต้องเมื่อ imm=0

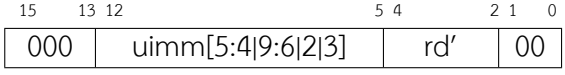


---

c.addi4spn rd′, uimm `x[8+rd′] = x[2] + uimm`

บวกค่าคงที่ ปรับขนาดด้วย 4 กับตัวชี้สแต็กโดยไม่ทำลายค่า RV32IC และ RV64IC.

ขยายเป็น **addi** rd, x2, uimm, โดยที่ rd=8+rd′. ไม่ถูกต้องเมื่อ uimm=0

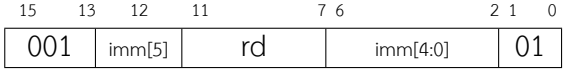


---

c.addiw rd, imm `x[rd] = sext((x[rd] + sext(imm))[31:0])`

บวกเวิร์ดค่าคงที่ *(Add Word Immediate)*. RV64IC เท่านั้น.

ขยายเป็น **addiw** rd, rd, imm. ไม่ถูกต้องเมื่อ rd=x0

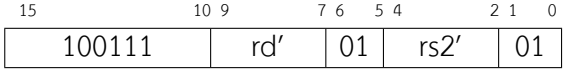


---

c.addw rd′, rs2′ `x[8+rd′] = sext((x[8+rd′] + x[8+rs2′])[31:0])`

บวกเวิร์ด *(Add Word)*. RV64IC เท่านั้น.

ขยายเป็น **addw** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′

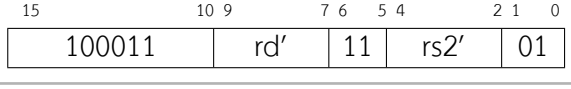


---

c.and rd′, rs2′ `x[8+rd′] = x[8+rd′] & x[8+rs2′]`

AND. RV32IC และ RV64IC.

ขยายเป็น **and** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′

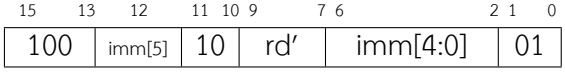


---

c.andi rd′, imm `x[8+rd′] = x[8+rd′] & sext(imm)`

AND ค่าคงที่ *AND Immediate*. RV32IC และ RV64IC.

ขยายเป็น **andi** rd, rd, imm, โดยที่ rd=8+rd′

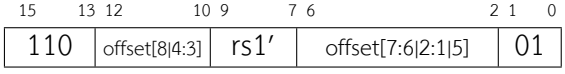


---

c.beqz rs1′, offset `if (x[8+rs1′] == 0) pc += sext(offset)`

กระโดดถ้าเท่ากับศูนย์ *Branch if Equal to Zero*. RV32IC และ RV64IC.

ขยายเป็น **beq** rs1, x0, offset, โดยที่ rs1=8+rs1′

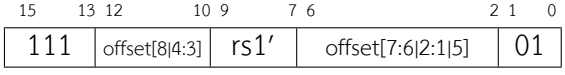


---

c.bnez rs1′, offset `if (x[8+rs1′] ≠ 0) pc += sext(offset)`

กระโดดถ้าไม่เท่ากับศูนย์ *Branch if Not Equal to Zero*. RV32IC และ RV64IC.

ขยายเป็น **bne** rs1, x0, offset, โดยที่ rs1=8+rs1′.

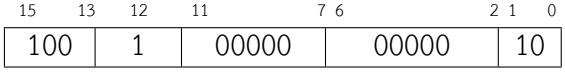


---

c.ebreak `RaiseException(Breakpoint)`

การหยุดสภาพแวดล้อม *(Environment Breakpoint)*. RV32IC และ RV64IC.

ขยายเป็น **ebreak**.

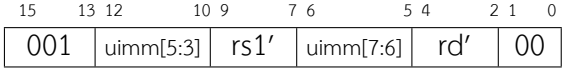


---

c.fld rd′, uimm(rs1′) `f[8+rd′] = M[x[8+rs1′] + uimm][63:0]`

โหลดทศนิยมดับเบิลเวิร์ด *(Floating-point Load Doubleword)*. RV32DC และ RV64DC.

ขยายเป็น **fld** rd, uimm(rs1), โดยที่ rd=8+rd′ และ rs1=8+rs1′



---

c.fldsp rd, uimm(x2) `f[rd] = M[x[2] + uimm][63:0]`

โหลดทศนิยมดับเบิลเวิร์ด,ตัวชี้สแต็กสัมพัทธ์ *(Floating-point Load Doubleword, Stack-Pointer Relative)*. RV32DC และ RV64DC.

ขยายเป็น **fld** rd, uimm(x2)

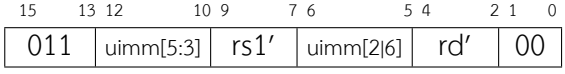


---

c.flw rd′, uimm(rs1′) `f[8+rd'] = M[x[8+rs1'] + uimm][31:0]`

โหลดทศนิยมเวิร์ด *(Floating-point Load Word)*. RV32FC เท่านั้น.

ขยายเป็น **flw** rd, uimm(rs1), โดยที่ rd=8+rd′ และ rs1=8+rs1′



---

c.flwsp rd, uimm(x2) `f[rd] = M[x[2] + uimm][31:0]`

โหลดทศนิยมเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Floating-point Load Word, Stack-Pointer Relative)*. RV32FC เท่านั้น.

ขยายเป็น **flw** rd, uimm(x2)

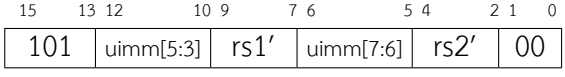


---

c.fsd rs2′, uimm(rs1′) `M[x[8+rs1'] + uimm][63:0] = f[8+rs2']`

บันทึกทศนิยมดับเบิลเวิร์ด *(Floating-point Store Doubleword)*. RV32DC และ RV64DC.

ขยายเป็น **fsd** rs2, uimm(rs1), where rs2=8+rs2′ และ rs1=8+rs1′

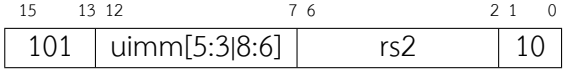


---

c.fsdsp rs2, uimm(x2) `M[x[2] + uimm][63:0] = f[rs2]`

บันทึกทศนิยมดับเบิลเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Floating-point Store Doubleword, Stack-Pointer Relative)*. RV32DC และ RV64DC.

ขยายเป็น **fsd** rs2, uimm(x2)

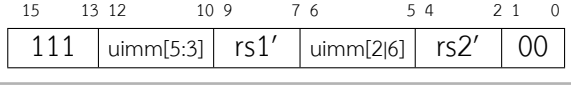


---

c.fsw rs2′, uimm(rs1′) `M[x[8+rs1'] + uimm][31:0] = f[8+rs2']`

บันทึกทศนิยมเวิร์ด *(Floating-point Store Word)*. RV32FC เท่านั้น.

ขยายเป็น **fsw** rs2, uimm(rs1), โดยที่ rs2=8+rs2′ และ rs1=8+rs1′



---

c.fswsp rs2, uimm(x2) `M[x[2] + uimm][31:0] = f[rs2]`

บันทึกทศนิยมเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Floating-point Store Word, Stack-Pointer Relative)*. RV32FC เท่านั้น.

ขยายเป็น **fsw** rs2, uimm(x2)

15–13	12–7	6–2	1–0
111	uimm[5:2|7:6]	rs2	10

---

c.j offset `pc += sext(offset)`

กระโดด *(Jump)*. RV32IC และ RV64IC.

ขยายเป็น **jal** x0, offset.

15–13	12–2	1–0
101	offset[11|4|9:8|10|6|7|3:1|5]	01

---

c.jal offset `x[1] = pc+2; pc += sext(offset)`

กระโดดและเชื่อมโยง *(Jump and Link)*. RV32IC เท่านั้น.

ขยายเป็น **jal** x1, offset.

15–13	12–2	1–0
001	offset[11|4|9:8|10|6|7|3:1|5]	01

---

c.jalr rs1 `t = pc+2; pc = x[rs1]; x[1] = t`

กระโดดและเชื่อมโยงรีจิสเตอร์ *(Jump and Link Register)*. RV32IC และ RV64IC.

ขยายเป็น **jalr** x1, 0(rs1). ไม่ถูกต้องเมื่อ rs1=x0

15–13	12	11–7	6–2	1–0
100	1	rs1	00000	10

---

c.jr rs1 `pc = x[rs1]`

รีจิสเตอร์กระโดด *(Jump Register)*. RV32IC และ RV64IC.

ขยายเป็น **jalr** x0, 0(rs1). ไม่ถูกต้องเมื่อ rs1=x0

15–13	12	11–7	6–2	1–0
100	0	rs1	00000	10

---

c.ld rd′, uimm(rs1′) `x[8+rd′] = M[x[8+rs1′] + uimm][63:0]`

โหลดดับเบิลเวิร์ด *(Load Doubleword)*. RV64IC เท่านั้น.

ขยายเป็น **ld** rd, uimm(rs1), โดยที่ rd=8+rd′ และ rs1=8+rs1′

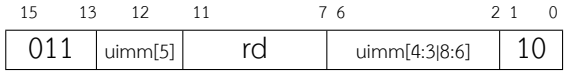
15–13	12–10	9–7	6–5	4–2	1–0
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00

---

c.ldsp rd, uimm(x2) `x[rd] = M[x[2] + uimm][63:0]`

โหลดดับเบิลเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Load Doubleword, Stack-Pointer Relative)*. RV64IC เท่านั้น.

ขยายเป็น **ld** rd, uimm(x2). ไม่ถูกต้องเมื่อ rd=x0



---

c.li rd, imm `x[rd] = sext(imm)`

โหลดค่าคงที่ *(Load Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **addi** rd, x0, imm



---

c.lui rd, imm `x[rd] = sext(imm[17:12] << 12)`

โหลดบิตบนค่าคงที่ *(Load Upper Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **lui** rd, imm. ไม่ถูกต้องเมื่อ rd=x2 or imm=0

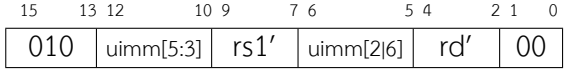


---

c.lw rd′, uimm(rs1′) `x[8+rd′] = sext(M[x[8+rs1′] + uimm][31:0])`

โหลดเวิร์ด *(Load Word)*. RV32IC และ RV64IC.

ขยายเป็น **lw** rd, uimm(rs1), โดยที่ rd=8+rd′ และ rs1=8+rs1′

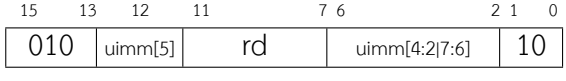


---

c.lwsp rd, uimm(x2) `x[rd] = sext(M[x[2] + uimm][31:0])`

โหลดเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Load Word, Stack-Pointer Relative)*. RV32IC และ RV64IC.

ขยายเป็น **lw** rd, uimm(x2). ไม่ถูกต้องเมื่อ rd=x0



---

c.mv rd, rs2 `x[rd] = x[rs2]`

คัดลอก *(Move)*. RV32IC และ RV64IC.

ขยายเป็น **add** rd, x0, rs2. ไม่ถูกต้องเมื่อ rs2=x0

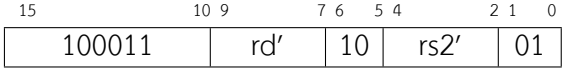


---

c.or rd′, rs2′ `x[8+rd′] = x[8+rd′] | x[8+rs2′]`

OR. RV32IC และ RV64IC.

ขยายเป็น **or** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′

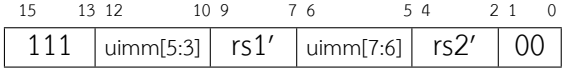


---

c.sd rs2′, uimm(rs1′) `M[x[8+rs1′] + uimm][63:0] = x[8+rs2′]`

บันทึกดับเบิลเวิร์ด *(Store Doubleword)*. RV64IC เท่านั้น.

ขยายเป็น **sd** rs2, uimm(rs1), โดยที่ rs2=8+rs2′ และ rs1=8+rs1′

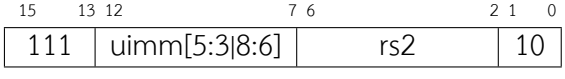


---

c.sdsp rs2, uimm(x2) `M[x[2] + uimm][63:0] = x[rs2]`

บันทึกดับเบิลเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Store Doubleword, Stack-Pointer Relative)*. RV64IC เท่านั้น.

ขยายเป็น **sd** rs2, uimm(x2)



---

c.slli rd, uimm `x[rd] = x[rd] << uimm`

เลื่อนบิตซ้ายค่าคงที่ *(Shift Left Logical Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **slli** rd, rd, uimm

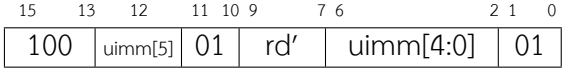


---

c.srai rd′, uimm `x[8+rd′] = x[8+rd′] >>s uimm`

เลื่อนบิตขวาค่าคงที่ *(Shift Right Arithmetic Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **srai** rd, rd, uimm, โดยที่ rd=8+rd′

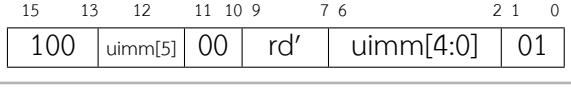


---

c.srli rd′, uimm `x[8+rd′] = x[8+rd′] >>u uimm`

เลื่อนบิตขวาค่าคงที่ *(Shift Right Logical Immediate)*. RV32IC และ RV64IC.

ขยายเป็น **srli** rd, rd, uimm, โดยที่ rd=8+rd′

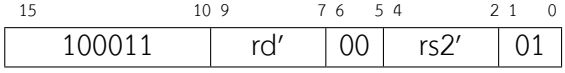


---

c.sub rd′, rs2′ `x[8+rd′] = x[8+rd′] - x[8+rs2′]`

ลบ *(Subtract)*. RV32IC และ RV64IC.

ขยายเป็น **sub** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′

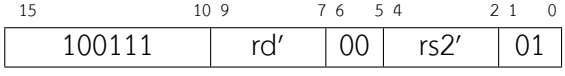


---

c.subw rd′, rs2′ `x[8+rd′] = sext((x[8+rd′] - x[8+rs2′])[31:0])`

ลบเวิร์ด *(Subtract Word)*. RV64IC เท่านั้น.

ขยายเป็น **subw** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′

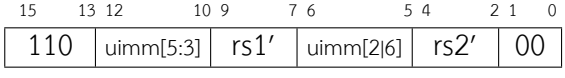


---

c.sw rs2′, uimm(rs1′) `M[x[8+rs1′] + uimm][31:0] = x[8+rs2′]`

บันทึกเวิร์ด *(Store Word)*. RV32IC และ RV64IC.

ขยายเป็น **sw** rs2, uimm(rs1), โดยที่ rs2=8+rs2′ และ rs1=8+rs1′

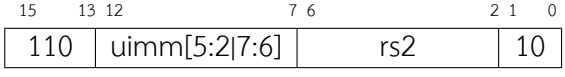


---

c.swsp rs2, uimm(x2) `M[x[2] + uimm][31:0] = x[rs2]`

บันทึกเวิร์ด, ตัวชี้สแต็กสัมพัทธ์ *(Store Word, Stack-Pointer Relative)*. RV32IC และ RV64IC.

ขยายเป็น **sw** rs2, uimm(x2)

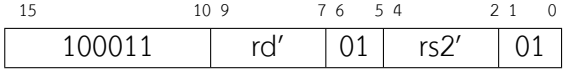


---

c.xor rd′, rs2′ `x[8+rd′] = x[8+rd′] ^ x[8+rs2′]`

เอ็กคลูซีฟ OR *(Exclusive-OR)*. RV32IC และ RV64IC.

ขยายเป็น **xor** rd, rd, rs2, โดยที่ rd=8+rd′ และ rs2=8+rs2′



---

call rd, symbol `x[rd] = pc+8; pc = &symbol`

เรียกฟังก์ชัน (Call). คำสั่งเทียม, RV32I และ RV64I.

เขียนตำแหน่งที่อยู่ของคำสั่งถัดไป (*pc*+8) ลงใน x[*rd*] จากนั้นกำหนด *pc* เป็น *symbol* คำสั่งนี้ขยายเป็น **auipc** rd, offsetHi แล้วตามด้วย **jalr** rd, offsetLo(rd) ถ้าไม่ระบุ *rd* จะถือว่าใช้ x1 โดยปริยาย

---

csrc csr, rs1 `CSRs[csr] &= ~x[rs1]`

ล้างรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Clear). คำสั่งเทียม, RV32I และRV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในรีจิสเตอร์ x[*rs1*] จะล้างบิตที่สอดคล้องกันในรีจิสเตอร์ควบคุมและสถานะ *CSR* ที่ระบุโดย csr โดยขยายเป็น **csrrc** x0, csr, rs1

---

csrci csr, zimm[4:0] `CSRs[csr] &= ~zimm`

ล้างรีจิสเตอร์ควบคุมและสถานะด้วยค่าคงที่ (Control and Status Register Clear Immediate). คำสั่งเทียม, RV32I และRV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในค่าคงที่ 5 บิตที่ขยายศูนย์ จะล้างบิตที่สอดคล้องกันในรีจิสเตอร์ควบคุมและสถานะ *CSR* ที่ระบุโดย csr โดยขยายเป็น **csrrci** x0, csr, zimm

---

csrr rd, csr `x[rd] = CSRs[csr]`

อ่านรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read). คำสั่งเทียม, RV32I และRV64I.

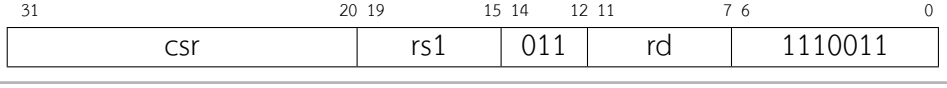
คัดลอกรีจิสเตอร์ควบคุมและสถานะ *csr* ไปที่ x[*rd*] โดยขยายเป็น **csrrs** rd, csr, x0

---

csrrc rd, csr, rs1 `t=CSRs[csr]; CSRs[csr] = t & ~x[rs1]; x[rd]=t`

อ่านค่าและล้างบิตรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Clear). รูปแบบ-I, RV32I และ RV64I.

ให้ *t* เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* เขียนผลลัพธ์ของการดำเนินการระดับบิต AND ระหว่าง *t* และส่วนเติมเต็มหนึ่งของรีจิสเตอร์ x[*rs1*] ไปยัง *csr* แล้วเขียนค่า *t* ลงใน x[*rd*]

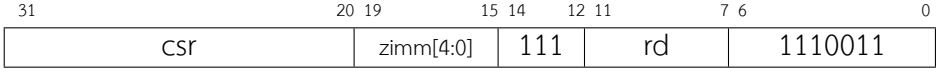


---

csrrci rd, csr, zimm[4:0] `t=CSRs[csr]; CSRs[csr] = t & ~zimm; x[rd]=t`

อ่านและล้างค่าคงที่รีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Clear Immediate). รูปแบบ-I, RV32I และ RV64I.

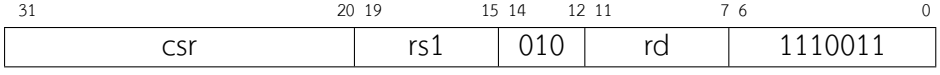
ให้ *t* เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* เขียนผลลัพธ์ของการดำเนินการระดับบิต AND ระหว่าง *t* และส่วนเติมเต็มหนึ่งของ*zimm* ค่าคงที่ 5 บิตขยายศูนย์ ไปยัง *csr* จากนั้นเขียนค่า *t* ไปยัง x[*rd*] (บิตที่ 5 ขึ้นไปใน *csr* จะไม่ถูกแก้ไข)



csrrs rd, csr, rs1 `t=CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd]=t`

อ่านและตั้งค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Set). รูปแบบ-I, RV32I และ RV64I.

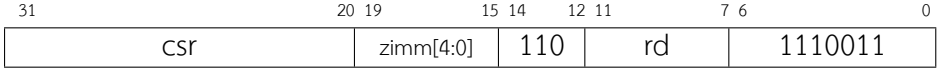
ให้ *t* เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* เขียนผลลัพธ์ของการดำเนินการระดับบิต OR ระหว่าง *t* และ x[*rs1*] ไปที่ *csr* จากนั้นเขียนค่า *t* ไปที่ x[*rd*]



csrrsi rd, csr, zimm[4:0] `t=CSRs[csr]; CSRs[csr] = t | zimm; x[rd]=t`

อ่านและตั้งค่าค่าคงที่รีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Set Immediate). รูปแบบ-I, RV32I และ RV64I.

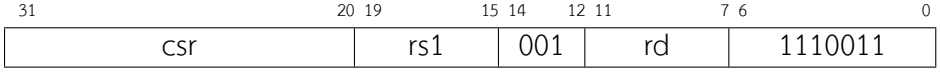
ให้ *t* เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* เขียนผลลัพธ์ของการดำเนินการระดับบิต OR ระหว่าง *t* และ ค่าคงที่ 5 บิตที่ขยายศูนย์ *zimm* ไปยัง *csr* จากนั้นเขียนค่า *t* ไปที่ x[*rd*] (บิตที่ 5 ขึ้นไปใน *csr* จะไม่ถูกแก้ไข)



csrrw rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t`

อ่านและเขียนค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Write). รูปแบบ-I, RV32I และ RV64I.

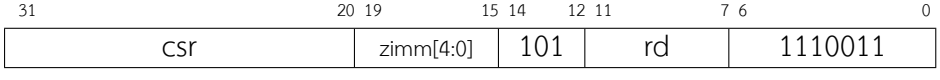
ให้ *t* เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* คัดลอก x[*rs1*] ไปยัง *csr* จากนั้นเขียนค่า *t* ไปยัง x[*rd*].



csrrwi rd, csr, zimm[4:0] `x[rd] = CSRs[csr]; CSRs[csr] = zimm`

อ่านและเขียนค่าคงที่รีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Write Immediate). รูปแบบ-I, RV32I และ RV64I.

คัดลอกค่าของรีจิสเตอร์ควบคุมและสถานะ *csr* ไปยัง x[*rd*] จากนั้นเขียนค่าคงที่ 5 บิตที่ขยายศูนย์ *zimm* ไปยัง *csr*



csrs csr, rs1 `CSRs[csr] |= x[rs1]`

ตั้งค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Set). คำสั่งเทียม, RV32I และRV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในรีจิสเตอร์ x[*rs1*] ให้ตั้งค่าบิตที่สอดคล้องในรีจิสเตอร์ควบคุมและสถานะ *csr* เป็น 1 ขยายเป็น **csrrs** x0, csr, rs1

---

csrsi csr, zimm[4:0] `CSRs[csr] |= zimm`

ตั้งค่าค่าคงที่รีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Set Immediate). คำสั่งเทียม, RV32I และRV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในค่าคงที่ 5 บิตที่ขยายศูนย์ ตั้งค่าบิตที่สอดคล้องในรีจิสเตอร์ตัวควบคุมและสถานะ *csr* เป็น 1 ขยายเป็น **csrrsi** x0, csr, zimm

---

csrw csr, rs1 `CSRs[csr] = x[rs1]`

เขียนรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Write). คำสั่งเทียม, RV32I และRV64I.

คัดลอก x[*rs1*] ไปยังรีจิสเตอร์ควบคุมและสถานะ *csr* ขยายเป็น **csrrw** x0, csr, rs1

---

csrwi csr, zimm[4:0] `CSRs[csr] = zimm`

เขียนค่าคงที่รีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Write Immediate). คำสั่งเทียม, RV32I และRV64I.

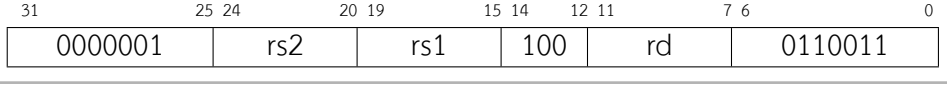
คัดลอก ค่าคงที่ 5 บิตที่ขยายศูนย์ ไปที่รีจิสเตอร์ควบคุมและสถานะ *csr* ขยายเป็น **csrrwi** x0, csr, zimm

---

div rd, rs1, rs2 `x[rd] = x[rs1] ÷s x[rs2]`

การหาร (Divide). รูปแบบ-R, RV32M และ RV64M.

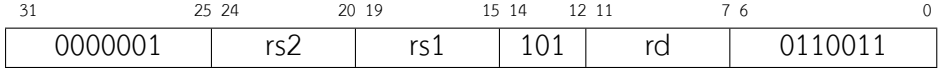
หารรีจิสเตอร์ x[*rs1*] ด้วยรีจิสเตอร์ x[*rs2*], ปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบส่วนเติมเต็มสองและเขียนผลหารไปยัง x[*rd*]



---

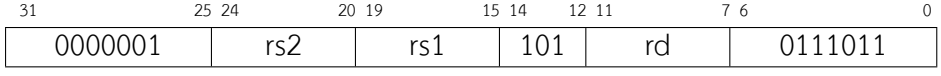
divu rd, rs1, rs2 `x[rd] = x[rs1] ÷u x[rs2]`

การหาร, ไม่มีเครื่องหมาย (Divide, Unsigned). รูปแบบ-R, RV32M และ RV64M.
หารรีจิสเตอร์ x[*rs1*] ด้วยรีจิสเตอร์ x[*rs2*] ปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียนผลหารไปยัง x[*rd*



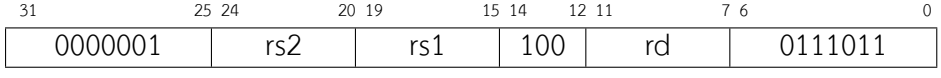
divuw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] ÷u x[rs2][31:0])`

การหารเวิร์ด, ไม่มีเครื่องหมาย (Divide Word, Unsigned). รูปแบบ-R, RV64M เท่านั้น.
หาร 32 บิตล่างของรีจิสเตอร์ x[*rs1*] ด้วย 32 บิตล่างของรีจิสเตอร์ x[*rs2*] ปัดเศษเข้าหาศูนย์ ถือว่าผลลัพท์เป็นจำนวนแบบไม่มีเครื่องหมาย และเขียนผลหาร 32 บิต แบบขยายเครื่องหมายไปยัง x[*rd*]



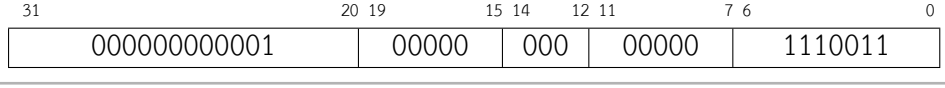
divw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] ÷s x[rs2][31:0])`

การหารเวิร์ด (Divide Word). รูปแบบ-R, RV64M เท่านั้น.
หารค่า 32 บิตล่างของรีจิสเตอร์ x[*rs1*] ด้วยค่า 32 บิตล่างของรีจิสเตอร์ x[*rs2*] ปัดเศษเข้าหาศูนย์ ถือว่าผลลัพท์เป็นจำนวนแบบส่วนเติมเต็มสองและเขียนผลหารแบบขยายครื่องหมายแบบ 32 บิต ไปยัง x[*rd*]



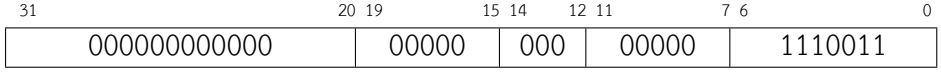
ebreak `RaiseException(Breakpoint)`

การหยุดสภาพแวดล้อม (Environment Breakpoint). รูปแบบ-I, RV32I และ RV64I.
สร้างคำร้องขอไปยังดีบักเกอร์โดยการเรียกข้อยกเว้นหยุดสภาพแวดล้อม



ecall `RaiseException(EnvironmentCall)`

การเรียกสภาพแวดล้อม (Environment Call). รูปแบบ-I, RV32I และ RV64I.
สร้างคำร้องขอไปยังสภาพแวดล้อมการประมวลผลโดยการเรียกข้อยกเว้นการเรียกสภาพแวดล้อม



fabs.d rd, rs1 `f[rd] = |f[rs1]|`

ค่าสัมบูรณ์เลขทศนิยมความแม่นยำสองเท่า (Floating-point Absolute Value). คำสั่งเทียม, RV32D และ RV64D.

เขียนค่าสัมบูรณ์ของตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ไปยัง f[*rd*] ขยายเป็น **fsgnjx.d** rd, rs1, rs1.

---

fabs.s rd, rs1 `f[rd] = |f[rs1]|`

ค่าสัมบูรณ์เลขทศนิยมความแม่นยำเท่าเดียว (Floating-point Absolute Value). คำสั่งเทียม, RV32F และRV64F.

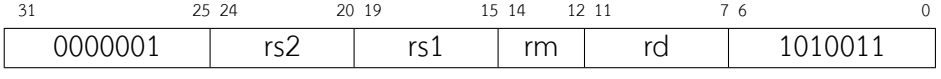
เขียนค่าสัมบูรณ์ของตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] ไปยัง f[*rd*] ขยายเป็น **fsgnjx.s** rd, rs1, rs1

---

fadd.d rd, rs1, rs2 `f[rd] = f[rs1] + f[rs2]`

บวกเลขทศนิยมความแม่นยำสองเท่า (Floating-point Add, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

บวกตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] และ f[*rs2*] และเขียนผลรวมแบบความแม่นยำสองเท่าที่ปัดเศษแล้วไปยัง f[*rd*].

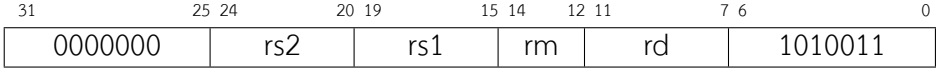


---

fadd.s rd, rs1, rs2 `f[rd] = f[rs1] + f[rs2]`

บวกเลขทศนิยมความแม่นยำเท่าเดียว (Floating-point Add, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

บวกตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] และ f[*rs2*] และเขียนผลรวมแบบความแม่นยำเท่าเดียวที่ปัดเศษแล้วไปยัง f[*rd*]

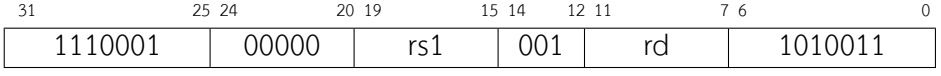


---

fclass.d rd, rs1 `x[rd] = classify`$_d$`(f[rs1])`

การจำแนกประเภททศนิยมแบบความแม่นยำสองเท่า (Floating-point Classify, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

เขียนมาสก์ที่ระบุประเภทของตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ไปยัง x[*rd*] ดูคำอธิบายของคำสั่ง **fclass.s** สำหรับการตีความค่าที่เขียนลงใน x[*rd*]



---

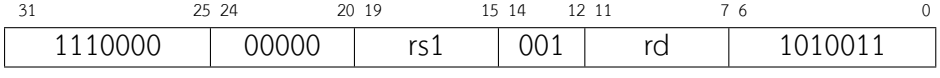
fclass.s rd, rs1

`x[rd] = classify`$_s$`(f[rs1])`

การจำแนกประเภททศนิยมแบบความแม่นยำเท่าเดียว (Floating-point Classify, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียนมาสก์ซึ่งระบุประเภทของตัวเลขทศนิยมแบบความแม่นยำเท่าเดียวใน f[*rs1*] ไปยัง x[*rd*] โดยจะมีเพียงบิตเดียวที่ตั้งค่าเป็น 1 ใน x[*rd*] ตามตารางต่อไปนี้:

[*rd*] bit	ความหมาย
0	f[*rs1*] จำนวนอนันต์ $-\infty$
1	f[*rs1*] จำนวนลบมาตรฐาน
2	f[*rs1*] จำนวนลบไม่มาตรฐาน
3	f[*rs1*] จำนวนอนันต์ลบ -0.
4	f[*rs1*] จำนวนอนันต์บวก $+0$.
5	f[*rs1*] จำนวนบวกไม่มาตรฐาน
6	f[*rs1*] จำนวนบวกมาตรฐาน
7	f[*rs1*] จำนวนอนันต์บวก $+\infty$
8	f[*rs1*] มีสัญญาณ NaN
9	f[*rs1*] ไม่มีสัญญาณ NaN

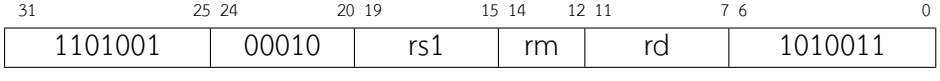


fcvt.d.l rd, rs1

`f[rd] = f64`$_{s64}$`(x[rs1])`

การแปลงจำนวนเต็ม 64 บิตเป็นเลขทศนิยมความแม่นยำสองเท่า (Floating-point Convert to Double from Long). รูปแบบ-R, RV64D เท่านั้น.

แปลงจำนวนเต็ม 64 บิตแบบส่วนเติมเต็มสองใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำสองเท่า และเขียนผลลัพท์ไปยัง f[*rd*]

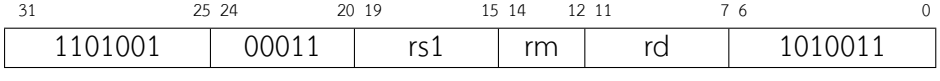


fcvt.d.lu rd, rs1

`f[rd] = f64`$_{u64}$`(x[rs1])`

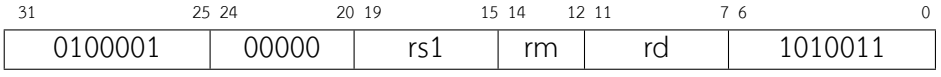
การแปลงจำนวนแต็ม 64 บิตแบบไม่มีเครื่องหมายเป็นเลขทศนิยมความแม่นยำสองเท่า (Floating-point Convert to Double from Unsigned Long). รูปแบบ-R, RV64D เท่านั้น.

แปลงจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมายใน x[*rs1*] เป็นเลขทศนิยมแบบความแม่นยำสองเท่า และเขียนไปยัง f[*rd*]



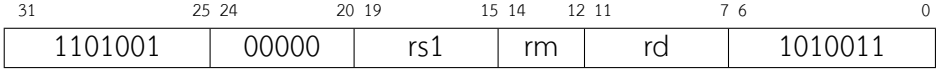
fcvt.d.s rd, rs1 `f[rd] = f64`$_{f32}$`(f[rs1])`

การแปลงเลขทศนิยมความแม่นยำเท่าเดียวเป็นทศนิยมความแม่นยำสองเท่า (Floating-point Convert to Double from Single). รูปแบบ-R, RV32D และ RV64D.
แปลงตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำสองเท่า และเขียนไปยัง f[*rd*]



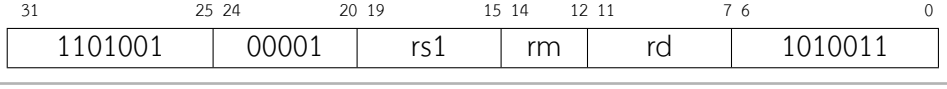
fcvt.d.w rd, rs1 `f[rd] = f64`$_{s32}$`(x[rs1])`

การแปลงเลขทศนิยมเป็นแบบความแม่นยำสองเท่าจากเวิร์ด (Floating-point Convert to Double from Word). รูปแบบ-R, RV32D และ RV64D.
แปลงจำนวนเต็ม 32 บิต แบบส่วนเติมเต็มสองในรีจิสเตอร์ x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำสองเท่า และเขียนไปยัง f[*rd*]



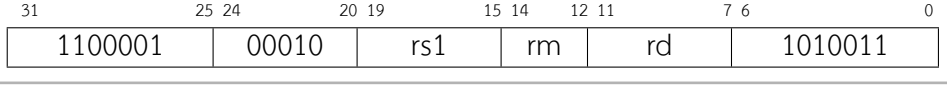
fcvt.d.wu rd, rs1 `f[rd] = f64`$_{u32}$`(x[rs1])`

การแปลงเลขทศนิยมเป็นทศนิยมแบบความแม่นยำสองเท่าจากเวิร์ดแบบไม่มีเครื่องหมาย (Floating-point Convert to Double from Unsigned Word). รูปแบบ-R, RV32D และ RV64D.
แปลงจำนวนเต็ม 32 บิต แบบไม่มีเครื่องหมาย ใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำสองเท่า และเขียนไปยัง f[*rd*]



fcvt.l.d rd, rs1 `x[rd] = s64`$_{f64}$`(f[rs1])`

การแปลงเลขทศนิยมเป็นจำนวนเต็ม 64 บิตจากความแม่นยำสองเท่า (Floating-point Convert to Long from Double). รูปแบบ-R, RV64D เท่านั้น.
แปลงตัวเลขทศนิยมที่มีความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 64 บิต แบบส่วนเติมเต็มสองเท่า และเขียนไปยัง x[*rd*]



fcvt.l.s rd, rs1 `x[rd] = s64`$_{f32}$`(f[rs1])`

การแปลงเลขทศนิยมความแม่นยำเท่าดียวเป็นจำนวนเต็ม 64 บิต (Floating-point Convert to Long from Single). รูปแบบ-R, RV64F เท่านั้น.

แปลงตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 64 บิต แบบส่วนเติมเต็มสอง และเขียนไปที่ x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00010	rs1	rm	rd	1010011

fcvt.lu.d rd, rs1 `x[rd] = u64`$_{f64}$`(f[rs1])`

การแปลงเลขทศนิยมความแม่นยำสองเท่าเป็นจำนวนเต็ม 64 บิตที่ไม่มีเครื่องหมาย (Floating-point Convert to Unsigned Long from Double). รูปแบบ-R, RV64D เท่านั้น.

แปลงตัวเลขทศนิยมที่มีความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 64 บิต แบบไม่มีเครื่องหมาย และเขียนไปที่ x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00011	rs1	rm	rd	1010011

fcvt.lu.s rd, rs1 `x[rd] = u64`$_{f32}$`(f[rs1])`

การแปลงเลขทศนิยมความแม่นยำเท่าเดียวเป็นจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย (Floating-point Convert to Unsigned Long from Single). รูปแบบ-R, RV64F เท่านั้น.

แปลงตัวเลขทศนิยมความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย และเขียนไปที่ x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	rs1	rm	rd	1010011

fcvt.s.d rd, rs1 `f[rd] = f32`$_{f64}$`(f[rs1])`

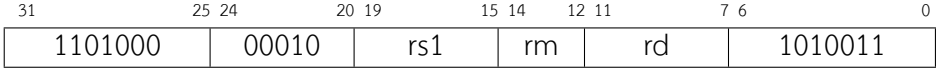
การแปลงเลขทศนิยมความแม่นยำสองเท่าเป็นความแม่นยำเท่าเดียว (Floating-point Convert to Single from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงตัวเลขทศนิยมที่มีความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว และเขียนไปที่ f[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
0100000	00001	rs1	rm	rd	1010011

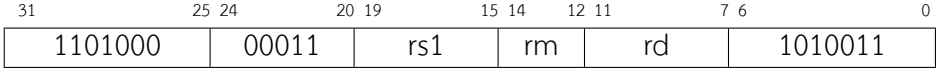
fcvt.s.l rd, rs1 `f[rd] = f32`$_{s64}$`(x[rs1])`

การแปลงเลขทศนิยมเป็นความแม่นยำเท่าเดียวจากจำนวนเต็ม 64 บิต (Floating-point Convert to Single from Long). รูปแบบ-R, RV64F เท่านั้น.
แปลงจำนวนเต็มส่วนเติมเต็มสอง64 บิต ใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว และเขียนไปที่ f[*rd*]



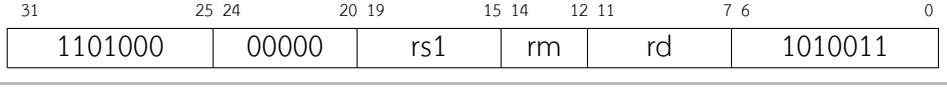
fcvt.s.lu rd, rs1 `f[rd] = f32`$_{u64}$`(x[rs1])`

การแปลงทศนิยมเป็นความแม่นยำเท่าเดียวจากจำนวนเต็ม 64 บิตที่ไม่มีเครื่องหมาย (Floating-point Convert to Single from Unsigned Long). รูปแบบ-R, RV64F เท่านั้น.
แปลงจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมายใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำเพียงเท่าเดียว และเขียนไปยัง f[*rd*]



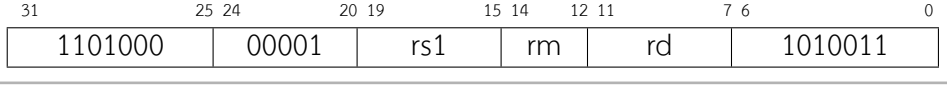
fcvt.s.w rd, rs1 `f[rd] = f32`$_{s32}$`(x[rs1])`

การแปลงเลขทศนิยมเป็นความแม่นยำเท่าเดียวจากเวิร์ด (Floating-point Convert to Single from Word). รูปแบบ-R, RV32F และ RV64F.
แปลงจำนวนเต็มแบบส่วนเติมเต็มสอง32 บิต ใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว และเขียนไปยัง f[*rd*]



fcvt.s.wu rd, rs1 `f[rd] = f32`$_{u32}$`(x[rs1])`

การแปลงทศนิยมเป็นความแม่นยำเท่าเดียวจากเวิร์ดที่ไม่มีเครื่องหมาย (Floating-point Convert to Single from Unsigned Word). รูปแบบ-R, RV32F และ RV64F.
แปลงจำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย ใน x[*rs1*] เป็นตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว และเขียนไปยัง f[*rd*]



fcvt.w.d rd, rs1 `x[rd] = sext(s32`$_{f64}$`(f[rs1]))`

การแปลงเลขทศนิยมความแม่นยำสองเท่าเป็นเวิร์ด (Floating-point Convert to Word from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 32 บิต แบบส่วนเติมเต็มสองและเขียนผลลัพธ์ที่ขยายเครื่องหมายไปยัง x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00000	rs1	rm	rd	1010011

fcvt.wu.d rd, rs1 `x[rd] = sext(u32`$_{f64}$`(f[rs1]))`

การแปลงจำนวนทศนิยมความแม่นยำสองเท่าเป็นเวิร์ดแบบไม่มีเครื่องหมาย (Floating-point Convert to Unsigned Word from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 32 บิต แบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ที่ขยายเครื่องหมายไปยัง x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00001	rs1	rm	rd	1010011

fcvt.w.s rd, rs1 `x[rd] = sext(s32`$_{f32}$`(f[rs1]))`

การแปลงเลขทศนิยมความแม่นยำเท่าเดียวเป็นเวิร์ด (Floating-point Convert to Word from Single). รูปแบบ-R, RV32F และ RV64F.

แปลงตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 32 บิต แบบส่วนเติมเต็มสองและเขียนผลลัพธ์ที่ขยายเครื่องหมายไปยัง x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00000	rs1	rm	rd	1010011

fcvt.wu.s rd, rs1 `x[rd] = sext(u32`$_{f32}$`(f[rs1]))`

การแปลงเลขทศนิยมความแม่นยำเท่าเดียวเป็นเวิร์ดแบบไม่มีเครื่องหมาย (Floating-point Convert to Unsigned Word from Single). รูปแบบ-R, RV32F และ RV64F.

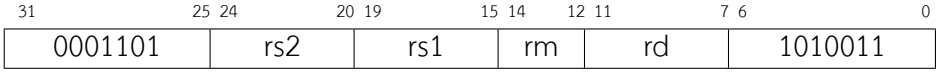
แปลงตัวเลขเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เป็นจำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ที่ขยายเครื่องหมาย ไปที่ x[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00001	rs1	rm	rd	1010011

fdiv.d rd, rs1, rs2 `f[rd] = f[rs1] ÷ f[rs2]`

การหารเลขทศนิยมความแม่นยำสองเท่า (Floating-point Divide, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

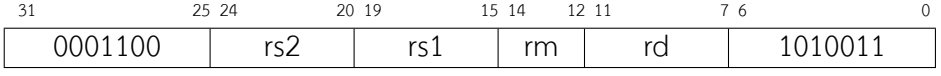
หารตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ด้วยรีจิสเตอร์ f[*rs2*] และเขียนผลหารที่ปัดเศษแบบความแม่นยำสองเท่าไปยัง f[*rd*]



fdiv.s rd, rs1, rs2 `f[rd] = f[rs1] ÷ f[rs2]`

การหารเลขทศนิยมความแม่นยำเท่าเดียว (Floating-point Divide, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

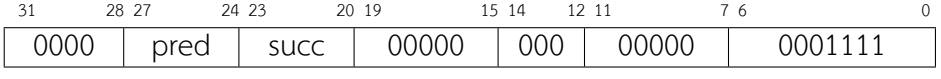
หารตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] ด้วยรีจิสเตอร์ f[*rs2*] และเขียนผลหารที่ปัดเศษแบบความแม่นยำเท่าเดียวไปยัง f[*rd*]



fence pred, succ `Fence(pred, succ)`

กำหนดลำดับหน่วยความจำและ I/O (Fence Memory and I/O). รูปแบบ-I, RV32I และ RV64I.

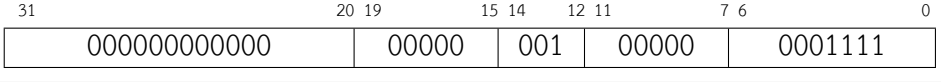
ทำให้การเข้าถึงหน่วยความจำและ I/O ก่อนหน้านี้ในชุด *pred*ecessor สามารถสังเกตได้โดยเธรดอื่นและอุปกรณ์ต่างๆ ก่อนที่การเข้าถึงหน่วยความจำและ I/O ถัดไปในชุด *succ*essor จะสามารถสังเกตได้ บิต 3, 2, 1 และ 0 ในชุดเหล่านี้สอดคล้องกับ: **i**nput ของอุปกรณ์, **o**utput ของอุปกรณ์, การ (**r**eads) หน่วยความจำ และการ **w**rites หน่วยความจำ ตามลำดับ ตัวอย่างเช่น คำสั่ง **fence** r,rw จะจัดลำดับการอ่านที่เก่ากว่า **กับ**การอ่านและการเขียนที่ใหม่กว่า และถูกเข้ารหัสด้วย: *pred*=0010 (หมายถึงควบคุมการอ่าน) และ *succ*=0011 (หมายถึงควบคุมการอ่านและการเขียน) หากไม่ระบุอาร์กิวเมนต์ จะถือว่าเป็น **fence** iorw, iorw แบบเต็ม



fence.i `Fence(Store, Fetch)`

กำหนดลำดับคำสั่งหน่วยความจำ (Fence Instruction Stream). รูปแบบ-I, RV32I และ RV64I.

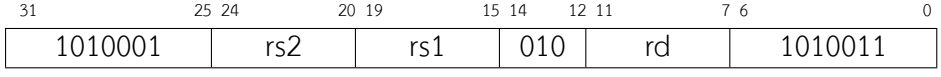
ทำให้การเก็บข้อมูลลงในหน่วยความจำคำสั่งสามารถสังเกตได้สำหรับการดึงคำสั่งที่ตามมา



feq.d rd, rs1, rs2 `x[rd] = f[rs1] == f[rs2]`

การเท่ากันของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Equals, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

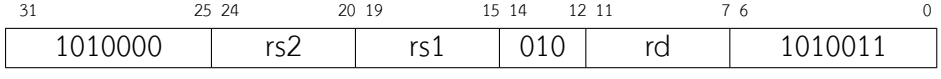
เขียน 1 ลงในรีจิสเตอร์ x[*rd*] ถ้าจำนวนทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] เท่ากับจำนวนจุดทศนิยม ในรีจิสเตอร์ f[*rs2*] และเขียน 0 ถ้าไม่เท่ากัน



feq.s rd, rs1, rs2 `x[rd] = f[rs1] == f[rs2]`

การเท่ากันของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Equals, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียน 1 ลงในรีจิสเตอร์ x[*rd*] ถ้าจำนวนทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] เท่ากับจำนวนจุดทศนิยม ในรีจิสเตอร์ f[*rs2*] และเขียน 0 ถ้าไม่เท่ากัน

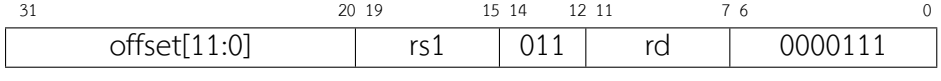


fld rd, offset(rs1) `f[rd] = M[x[rs1] + sext(offset)][63:0]`

โหลดจุดทศนิยมดับเบิลเวิร์ด (Floating-point Load Doubleword). รูปแบบ-I, RV32D และ RV64D.

โหลดจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า จากตำแหน่งที่อยู่หน่วยความจำ x[*rs1*] *ขยายเครื่องหมาย*(*offset*) และเขียนไปยัง f[*rd*].

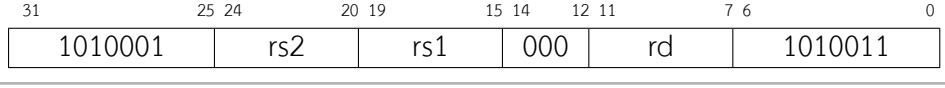
คำสั่งบีบอัด: **c.fldsp** rd, offset; **c.fld** rd, offset(rs1)



fle.d rd, rs1, rs2 `x[rd] = f[rs1] ≤ f[rs2]`

การน้อยกว่าหรือเท่ากับของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Less Than or Equal, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

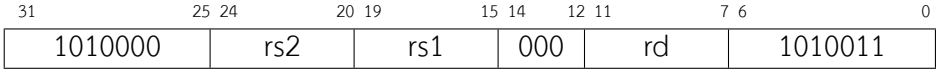
เขียน 1 ไปที่รีจิสเตอร์ x[*rd*] ถ้าจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] มีค่าน้อยกว่าหรือเท่ากับจำนวนจุดทศนิยม ในรีจิสเตอร์ f[*rs2*] และถ้าไม่ใช่ เขียน 0



fle.s rd, rs1, rs2 `x[rd] = f[rs1] ≤ f[rs2]`

การน้อยกว่าหรือเท่ากับของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Less Than or Equal, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

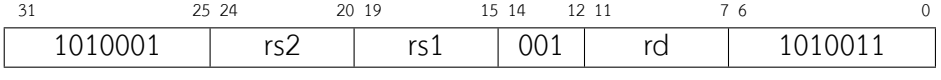
เขียน 1 ไปที่รีจิสเตอร์ x[*rd*] ถ้าจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] มีค่าน้อยกว่าหรือเท่ากับจำนวนจุดทศนิยม ในรีจิสเตอร์ f[*rs2*] และถ้า ไม่ใช่ เขียน 0



flt.d rd, rs1, rs2 `x[rd] = f[rs1] < f[rs2]`

การน้อยกว่าของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Less Than, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

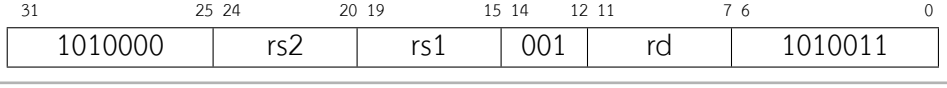
เขียน 1 ไปที่รีจิสเตอร์ x[*rd*] ถ้าจำนวนจุดทศนิยมแบบความแม่นสองเท่า ในรีจิสเตอร์ f[*rs1*] มีค่าน้อยกว่าจำนวนจุดทศนิยม ในรีจิสเตอร์ f[*rs2*] และถ้าไม่ใช่ เขียน 0



flt.s rd, rs1, rs2 `x[rd] = f[rs1] < f[rs2]`

การน้อยกว่าของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Less Than, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียน 1 ไปที่รีจิสเตอร์ x[*rd*] ถ้าจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] มีค่าน้อยกว่าจำนวนจุดทศนิยม ใน f[*rs2*] และถ้าไม่ใช่ เขียน 0

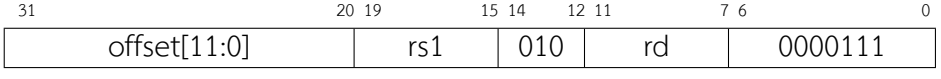


flw rd, offset(rs1) `f[rd] = M[x[rs1] + sext(offset)][31:0]`

การโหลดจุดทศนิยมแบบเวิร์ด (Floating-point Load Word). รูปแบบ-I, RV32F และ RV64F.

โหลดจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว จากตำแหน่งที่อยู่หน่วยความจำ x[*rs1*] + *sign-extend*(*offset*) และเขียนไปยังรีจิสเตอร์ f[*rd*].

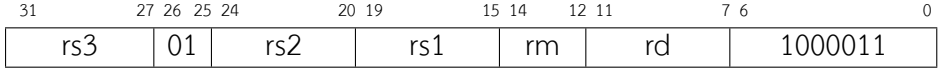
คำสั่งบีบอัด: **c.flwsp** rd, offset; **c.flw** rd, offset(rs1)



fmadd.d rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]+f[rs3]`

การคูณ-บวกแบบผสมของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Fused Multiply-Add, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.

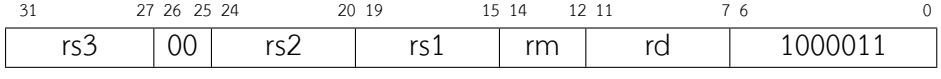
คูณจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ใน f[*rs1*] กับ f[*rs2*] บวกผลคูณที่ยังไม่ถูกปัดเศษเข้ากับจำนวนจุดทศนิยมความแม่นยำสองเท่า ใน f[*rs3*] และเขียนผลลัพธ์แบบความแม่นยำสองเท่าที่ถูกปัดเศษแล้ว ไปยังรีจิสเตอร์ f[*rd*]



fmadd.s rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]+f[rs3]`

การคูณ-บวกแบบผสมของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Fused Multiply-Add, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.

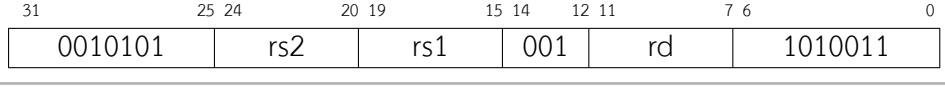
คูณจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ใน f[*rs1*] กับ f[*rs2*] บวกผลคูณที่ไม่ยังถูกปัดเศษ กับจำนวนจุดทศนิยมความแม่นยำเท่าเดียว ใน f[*rs3*] และเขียนผลลัพธ์แบบความแม่นยำเท่าเดียวแบบที่ปัดเศษแล้ว ไปยังรีจิสเตอร์ f[*rd*]



fmax.d rd, rs1, rs2 `f[rd] = max(f[rs1], f[rs2])`

การหาค่าสูงสุดของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Maximum, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

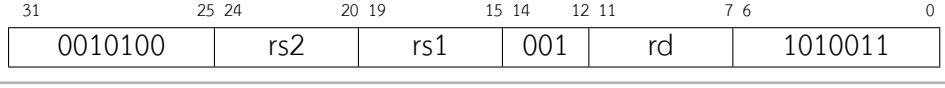
คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ที่มีค่ามากกว่าระหว่างรีจิสเตอร์ f[*rs1*] และ f[*rs2*] ไปยังรีจิสเตอร์ f[*rd*]



fmax.s rd, rs1, rs2 `f[rd] = max(f[rs1], f[rs2])`

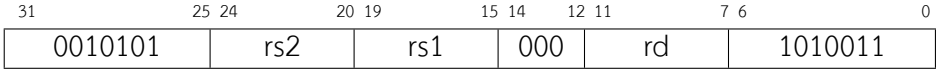
การหาค่าสูงสุดของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Maximum, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียวที่มีค่ามากกว่า ระหว่างรีจิสเตอร์ f[*rs1*] และ f[*rs2*] ไปยังรีจิสเตอร์ f[*rd*]



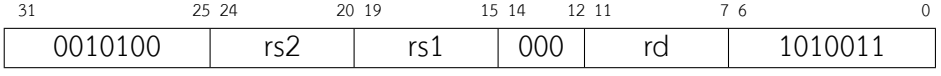
fmin.d rd, rs1, rs2 `f[rd] = min(f[rs1], f[rs2])`

การหาค่าต่ำสุดของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Minimum, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ที่มีค่าน้อยกว่า ระหว่างรีจิสเตอร์ f[*rs1*] และ f[*rs2*] ไปยังรีจิสเตอร์ f[*rd*]



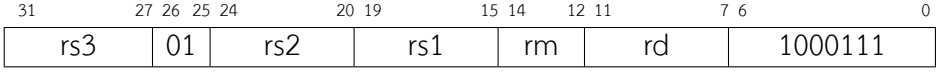
fmin.s rd, rs1, rs2 `f[rd] = min(f[rs1], f[rs2])`

การหาค่าต่ำสุดของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Minimum, Single-Precision). รูปแบบ-R, RV32F และ RV64F.
คัดลอกจำนวนจุดทศนิยมความแม่นยำเท่าเดียว ที่มีค่าน้อยกว่า ระหว่างรีจิสเตอร์ f[*rs1*] และ f[*rs2*] ไปยังรีจิสเตอร์ f[*rd*]



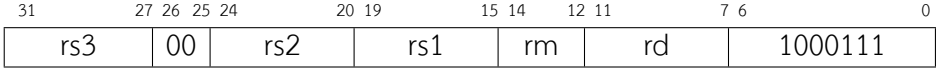
fmsub.d rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]−f[rs3]`

การคูณ-ลบแบบผสมของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Fused Multiply-Subtract, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.
คูณจำนวนจุดทศนิยมที่มีความแม่นยำสองเท่า ระหว่างรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] แล้วลบจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs3*] จากผลคูณที่ยังไม่ถูกปัดเศษ และเขียนผลลัพธ์แบบความแม่นยำสองเท่าที่ปัดเศษแล้วไปยัง f[*rd*]



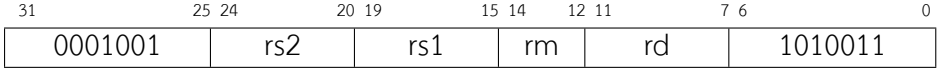
fmsub.s rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]−f[rs3]`

การคูณ-ลบแบบผสมของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Fused Multiply-Subtract, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.
คูณตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ระหว่างรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] ลบตัวเลขทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs3*] จากผลคูณที่ไม่ถูกปัดเศษ และเขียนผลลัพธ์แบบความแม่นยำเดียวที่ปัดเศษแล้วไปยัง f[*rd*]



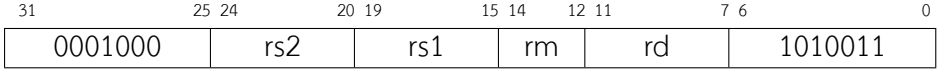
fmul.d rd, rs1, rs2 `f[rd] = f[rs1] × f[rs2]`

การคูณจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Multiply, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
คูณจำนวนจุดลอยทศนิยมแบบความแม่นยำสองเท่า ระหว่างรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] และเขียนผลคูณแบบความแม่นยำสองเท่าไปยัง f[*rd*]



fmul.s rd, rs1, rs2 `f[rd] = f[rs1] × f[rs2]`

การคูณจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Multiply, Single-Precision). รูปแบบ-R, RV32F และ RV64F.
คูณจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ระหว่างรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] และเขียนผลคูณความแม่นยำเท่าเดียวไปยัง f[*rd*]

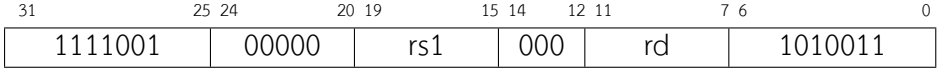


fmv.d rd, rs1 `f[rd] = f[rs1]`

การย้ายจำนวนจุดทศนิยม (Floating-point Move). คำสั่งเทียม, RV32D และ RV64D.
คัดลอกเลขทศนิยมความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ไปยัง f[*rd*] ขยายเป็น **fsgnj.d** rd, rs1, rs1

fmv.d.x rd, rs1 `f[rd] = x[rs1][63:0]`

การย้ายจุดทศนิยมแบบความแม่นยำสองเท่า (Floating-point Move Doubleword from Integer). รูปแบบ-R, RV64D เท่านั้น.
คัดลอกจำนวนจุดตัวเลขทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ x[*rs1*] ไปยัง f[*rd*]



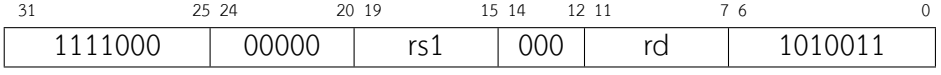
fmv.s rd, rs1 `f[rd] = f[rs1]`

การย้ายจำนวนจุดทศนิยม (Floating-point Move). คำสั่งเทียม, RV32F และ RV64F.
คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] ไปยัง f[*rd*] ขยายเป็น **fsgnj.d** rd, rs1, rs1

fmv.w.x rd, rs1 `f[rd] = x[rs1][31:0]`

การย้ายเลขทศนิยมแบบเวิร์ดจากจำนวนเต็ม (Floating-point Move Word from Integer). รูปแบบ-R, RV32F และ RV64F.

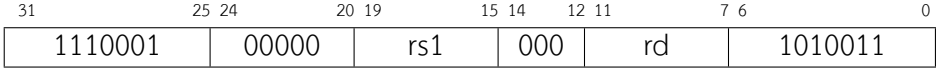
คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ x[*rs1*] ไปยัง f[*rd*]



fmv.x.d rd, rs1 `x[rd] = f[rs1][63:0]`

การย้ายเลขทศนิยมดับเบิลเวิร์ดเป็นจำนวนเต็ม (Floating-point Move Doubleword to Integer). รูปแบบ-R, RV64D เท่านั้น.

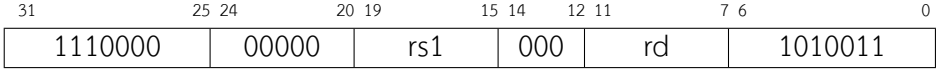
คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ไปยัง x[*rd*]



fmv.x.w rd, rs1 `x[rd] = sext(f[rs1][31:0])`

การย้ายเลขทศนิยมแบบเวิร์ดเป็นจำนวนเต็ม (Floating-point Move Word to Integer). รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] ไปยัง x[*rd*] โดยทำการขยายเครื่องหมายของผลลัพธ์สำหรับ RV64F



fneg.d rd, rs1 `f[rd] = -f[rs1]`

การเปลี่ยนเครื่องหมายของเลขทศนิยม (Floating-point Negate). คำสั่งเทียม, RV32D และ RV64D.

เขียนค่าตรงข้ามของบิตเครื่องหมายของจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] ลงใน f[*rd*] ขยายเป็น **fsgnjn.d** rd, rs1, rs1

fneg.s rd, rs1 `f[rd] = -f[rs1]`

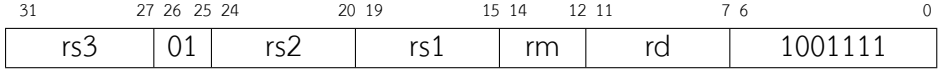
การเปลี่ยนเครื่องหมายของจุดทศนิยม (Floating-point Negate). คำสั่งเทียม, RV32D และ RV64D.

เขียนค่าตรงข้ามของบิตเครื่องหมายของจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] ลงใน f[*rd*] ขยายเป็น **fsgnjn.s** rd, rs1, rs1

fnmadd.d rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]-f[rs3]`

การคูณ-บวกเชิงลบแบบผสมของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Fused Negative Multiply-Add, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.

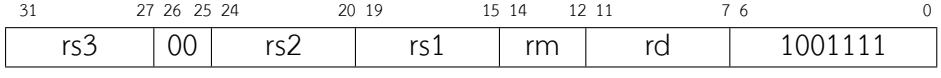
คูณจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] เปลี่ยนผลลัพธ์ให้เป็นค่าลบ ลบจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs3*] ออกจากผลคูณที่ยังไม่ถูกปัดเศษ และเขียนผลลัพธ์แบบความแม่นยำสองเท่าที่ถูกปัดเศษแล้วลงใน f[*rd*]



fnmadd.s rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]-f[rs3]`

การคูณ-บวกเชิงลบแบบแบบผสมของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Fused Negative Multiply-Add, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.

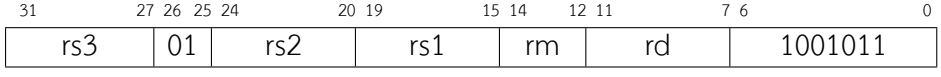
คูณจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] เปลี่ยนผลลัพธ์ให้เป็นค่าลบ ลบจำนวนจุดทศนิยมแบบความแม่นยำเดียว ในรีจิสเตอร์ f[*rs3*] ออกจากผลคูณที่ยังไม่ถูกปัดเศษ และเขียนผลลัพธ์แบบความแม่นยำเดียวที่ถูกปัดเศษแล้วลงใน f[*rd*]



fnmsub.d rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]+f[rs3]`

การคูณ-ลบเชิงลบแบบผสมของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Fused Negative Multiply-Subtract, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.

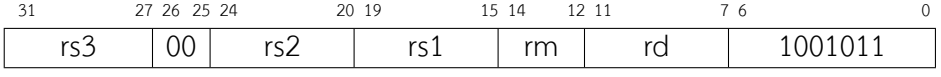
คูณจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] เปลี่ยนผลลัพธ์ให้เป็นค่าลบ บวกผลคูณที่ยังไม่ถูกปัดเศษกับจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ใน f[*rs3*] และเขียนผลลัพธ์แบบความแม่นยำสองเท่าที่ถูกปัดเศษแล้วลงใน f[*rd*]



fnmsub.s rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]+f[rs3]`

การคูณ-ลบเชิงลบแบบผสมของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Fused Negative Multiply-Subtract, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.

คูณจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียวในรีจิสเตอร์ f[*rs1*] กับ f[*rs2*] เปลี่ยนผลลัพธ์ให้เป็นค่าลบ บวกผลคูณที่ยังไม่ปัดเศษเข้ากับจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs3*] และเขียนผลลัพธ์แบบความแม่นยำเท่าเดียวที่ถูกปัดเศษแล้วลงใน f[*rd*]



---

frcsr rd `x[rd] = CSRs[fcsr]`

การอ่านรีจิสเตอร์ควบคุมและสถานะทศนิยม (Floating-point Read Control and Status Register). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกรีจิสเตอร์ควบคุมและสถานะทศนิยม ไปยัง x[*rd*] ขยายเป็น **csrrs** rd, fcsr, x0

---

frflags rd `x[rd] = CSRs[fflags]`

การอ่านแฟล็กข้อยกเว้นทศนิยม (Floating-point Read Exception Flags). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกแฟล็กข้อยกเว้นทศนิยมไปยัง x[*rd*] ขยายเป็น **csrrs** rd,fflags, x0

---

frrm rd `x[rd] = CSRs[frm]`

การอ่านโหมดการปัดเศษทศนิยม (Floating-point Read Rounding Mode). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกโหมดการปัดเศษทศนิยมไปยัง x[*rd*] ขยายเป็น **csrrs** rd, frm, x0

---

fscsr rd, rs1 `t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t`

การสลับรีจิสเตอร์การควบคุมและสถานะทศนิยม (Floating-point Swap Control and Status Register). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกรีจิสเตอร์ x[*rs1*] ไปยังรีจิสเตอร์ควบคุมและสถานะ จากนั้นคัดลอกค่าก่อนหน้าของรีจิสเตอร์ควบคุมและสถานะทศนิยม ไปยัง x[*rd*] ขยายเป็น **csrrw** rd, fcsr, rs1 ถ้าไม่ระบุ *rd* จะถือว่าเป็น x0

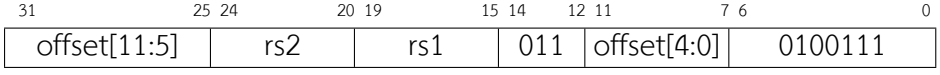
---

fsd rs2, offset(rs1) `M[x[rs1] + sext(offset)] = f[rs2][63:0]`

การบันทึกจุดทศนิยมความแม่นยำสองเท่า (Floating-point Store Doubleword). รูปแบบ-S, RV32D และ RV64D.

บันทึกจำนวนจุดทศนิยมความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + *sign-extend*(*offset*)

คำสั่งบีบอัด: **c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)



---

fsflags rd, rs1 `t=CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd]=t`

การสลับแฟล็กข้อยกเว้นทศนิยม (Floating-point Swap Exception Flags). คำสั่งเทียม, RV32F และ RV64F.

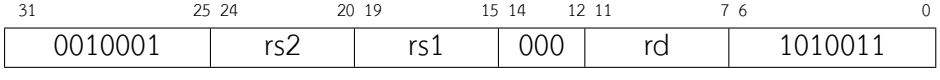
คัดลอกรีจิสเตอร์ x[*rs1*] ไปยังรีจิสเตอร์แฟล็กข้อยกเว้นทศนิยม จากนั้นคัดลอกค่าก่อนหน้าของแฟล็กข้อยกเว้นทศนิยมไปที่ x[*rd*] ขยายเป็น **csrrw** rd,fflags, rs1 ถ้า *rd* ถูกละไว้จะถือว่าเป็น x0

---

fsgnj.d rd, rs1, rs2 `f[rd] = {f[rs2][63], f[rs1][62:0]}`

การฉีดเครื่องหมายของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Sign Inject, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

สร้างจำนวนจุดทศนิยมแบบความแม่นยำสองเท่าใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] ด้วยบิตเครื่องหมายจาก f[*rs2*] และเขียนลงใน f[*rd*]

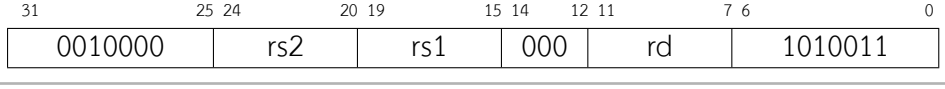


---

fsgnj.s rd, rs1, rs2 `f[rd] = {f[rs2][31], f[rs1][30:0]}`

การฉีดเครื่องหมายของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Sign Inject, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

สร้างจำนวนจุดทศนิยมแบบความแม่นยำเดียวใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] ด้วยบิตเครื่องหมายจาก f[*rs2*] และเขียนลงใน f[*rd*]



---

fsgnjn.d rd, rs1, rs2 `f[rd] = {~f[rs2][63], f[rs1][62:0]}`

การฉีดเครื่องหมายเชิงลบของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Sign Inject-Negate, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
สร้างจำนวนจุดทศนิยมความแม่นยำสองเท่าใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] โดยใช้เครื่องหมายตรงข้ามของ f[*rs2*] และเขียนลงใน f[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	001	rd	1010011

fsgnjn.s rd, rs1, rs2 `f[rd] = {~f[rs2][31], f[rs1][30:0]}`

การฉีดเครื่องหมายเชิงลบของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Sign Inject-Negate, Single-Precision). รูปแบบ-R, RV32F และ RV64F.
สร้างจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียวใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] โดยทำการสลับค่าบิตเครื่องหมายใน f[*rs2*] และเขียนลงใน f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	001	rd	1010011

fsgnjx.d rd, rs1, rs2 `f[rd]={f[rs1][63] ^ f[rs2][63],f[rs1][62:0]}`

การฉีดเครื่องหมายแบบ XOR ของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Sign Inject-XOR, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
สร้างจำนวนจุดทศนิยมแบบความแม่นยำสองเท่าใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] โดยใช้เครื่องหมายที่ได้จากการทำ XOR ระหว่างเครื่องหมายของ f[*rs1*] และ f[*rs2*] และเขียนลงใน f[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	010	rd	1010011

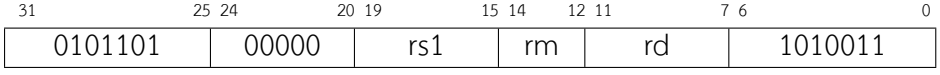
fsgnjx.s rd, rs1, rs2 `f[rd]={f[rs1][31] ^ f[rs2][31],f[rs1][30:0]}`

การฉีดเครื่องหมายแบบ XOR ของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Sign Inject-XOR, Single-Precision). รูปแบบ-R, RV32F และ RV64F.
สร้างจำนวนจุดทศนิยมความแม่นยำเท่าเดียวใหม่ จากเลขชี้กำลังและเลขนัยสำคัญของ f[*rs1*] โดยใช้เครื่องหมายที่ได้จากการทำ XOR ระหว่างเครื่องหมายของ f[*rs1*] และ f[*rs2*] และเขียนลงใน f[*rd*]

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	010	rd	1010011

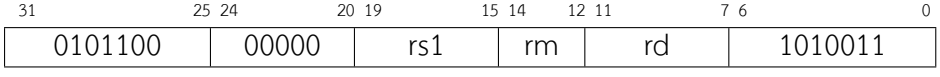
fsqrt.d rd, rs1 `f[rd] = √f[rs1]`

รากที่สองของจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Square Root, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
คำนวณรากที่สองของจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ใน รีจิสเตอร์ f[*rs1*] และเขียนผลลัพธ์ความแม่นยำสองเท่าที่ถูกปัดเศษแล้วลงใน f[*rd*]



fsqrt.s rd, rs1 `f[rd] = √f[rs1]`

รากที่สองของจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Square Root, Single-Precision). รูปแบบ-R, RV32F และ RV64F.
คำนวณรากที่สองของจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs1*] และเขียนผลลัพธ์ความแม่นยำเท่าเดียวที่ถูกปัดเศษแล้วลงใน f[*rd*]

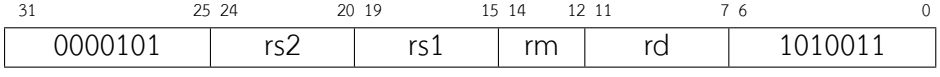


fsrm rd, rs1 `t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t`

การสลับโหมดการปัดเศษทศนิยม (Floating-point Swap Rounding Mode). คำสั่งเทียม, RV32F และ RV64F.
คัดลอก x[*rs1*] ไปยังรีจิสเตอร์โหมดการปัดเศษทศนิยม จากนั้นคัดลอกโหมดการปัดเศษทศนิยมก่อนหน้าไปยัง x[*rd*] ขยายเป็น **csrrw** rd, frm, rs1 ถ้าไม่ระบุ *rd* จะถือว่าเป็น x0

fsub.d rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

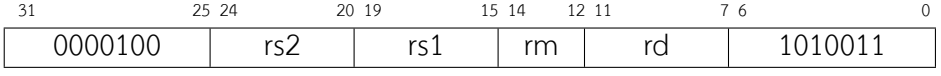
การลบจุดทศนิยม, ความแม่นยำสองเท่า (Floating-point Subtract, Double-Precision). รูปแบบ-R, RV32D และ RV64D.
ลบจำนวนจุดทศนิยมแบบความแม่นยำสองเท่า ในรีจิสเตอร์ f[*rs2*] ออกจาก f[*rs1*] และเขียนผลลัพธ์ของผลต่างแบบความแม่นยำสองเท่าที่ถูกปัดเศษแล้วลงใน f[*rd*]



fsub.s rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

การลบจุดทศนิยม, ความแม่นยำเท่าเดียว (Floating-point Subtract, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

ลบจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs2*] ออกจาก f[*rs1*] และเขียนผลลัพธ์ของผลต่างแบบความแม่นยำเท่าเดียวที่ถูกปัดเศษแล้วลงใน f[*rd*]



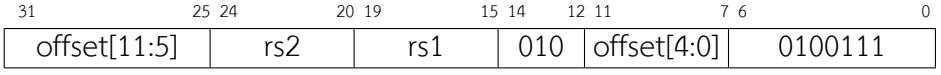
---

fsw rs2, offset(rs1) `M[x[rs1] + sext(offset)]= f[rs2][31:0]`

การจัดเก็บจุดทศนิยมแบบเวิร์ด (Floating-point Store Word). รูปแบบ-S, RV32F และ RV64F.

บันทึกจำนวนจุดทศนิยมแบบความแม่นยำเท่าเดียว ในรีจิสเตอร์ f[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + *sign-extend*(*offset*)

คำสั่งปีบอัด: **c.fswsp** rs2, offset; **c.fsw** rs2, offset(rs1)



---

j offset `pc += sext(offset)`

กระโดด (Jump). คำสั่งเทียม, RV32I และ RV64I.

ตั้งค่า *pc* เป็นค่า *pc* ปัจจุบัน บวกกับส่วนขยายเครื่องหมาย *offset* ขยายเป็น **jal** x0, offset

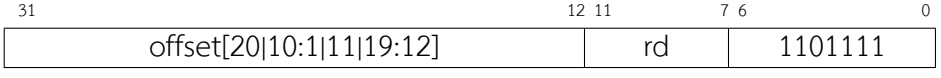
---

jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

การกระโดดและเชื่อมโยง (Jump and Link). รูปแบบ-J, RV32I และ RV64I.

เขียนตำแหน่งที่อยู่ของคำสั่งถัดไป (*pc*+4) ลงใน x[*rd*] จากนั้นตั้งค่า *pc* เป็น *pc* ปัจจุบันบวกกับ *offset* ที่ถูกขยายเครื่องหมาย ถ้าไม่ระบุ *rd* จะถือว่าเป็น x1

คำสั่งปีบอัด: **c.j** offset; **c.jal** offset



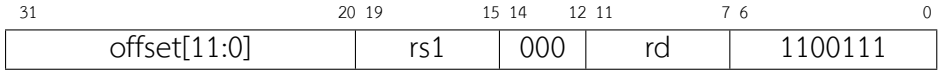
---

jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t`

กระโดดและเชื่อมโยงผ่านรีจิสเตอร์ (Jump and Link Register). รูปแบบ-I, RV32I และ RV64I.

ตั้งค่า *pc* เป็น x[*rs1*]+*sign-extend* (*offset*) โดยตัดบิตที่มีนัยสำคัญน้อยที่สุดของตำแหน่งที่อยู่ที่คำนวณได้ออก จากนั้นเขียนค่าก่อนหน้าของ *pc*+4 ลงใน x[*rd*] ถ้าไม่ระบุ *rd* จะถือว่าเป็น x1

คำสั่งบีบอัด: **c.jr** rs1; **c.jalr** rs1



---

jr rs1 `pc = x[rs1]`

การกระโดดผ่านเรจิสเตอร์ (Jump Register). คำสั่งเทียม, RV32I และRV64I.

ตั้งค่า *pc* เป็น x[*rs1*] ขยายเป็น **jalr** x0, 0(rs1)

---

la rd, symbol `x[rd] = &symbol`

โหลดที่อยู่ (Load Address). คำสั่งเทียม, RV32I และ RV64I.

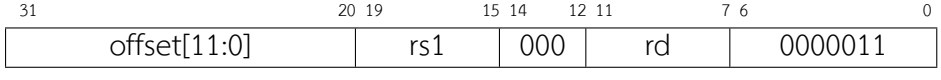
โหลดที่อยู่ของ สัญลักษณ์ *(symbo)* ลงใน x[*rd*] เมื่อทำการแอสเซมบลีโค้ดแบบไม่ขึ้นกับตำแหน่ง คำสั่งนี้จะถูกขยายเป็นการโหลดจากตารางออฟเซ็ตสากล (Global Offset Table): สำหรับ RV32I, **auipc** rd, offsetHi จากนั้น **lw** rd, offsetLo(rd); สำหรับ RV64I, **auipc** rd, offsetHi จากนั้น **ld** rd, offsetLo(rd) ในกรณีอื่น คำสั่งจะถูกขยายเป็น **auipc** rd, offsetHi จากนั้น **addi** rd, rd, offsetLo

---

lb rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][7:0])`

โหลดไบต์ (Load Byte). รูปแบบ-I, RV32I และ RV64I.

โหลดไบต์จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + (*offset*) *ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*] โดยทำการขยายเครื่องหมายให้กับผลลัพท์

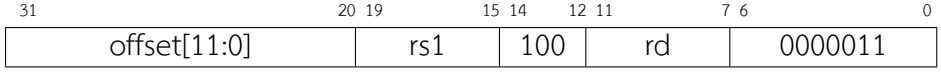


---

lbu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][7:0]`

โหลดไบต์, ไม่มีเครื่องหมาย (Load Byte, Unsigned). รูปแบบ-I, RV32I และ RV64I.

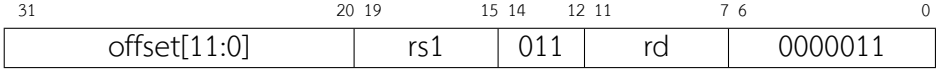
โหลดไบต์จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + (*offset ที่ขยายเครื่องหมาย*) และเขียนลงใน x[*rd*] โดยทำการขยายศูนย์ให้กับผลลัพท์



---

ld rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][63:0]`

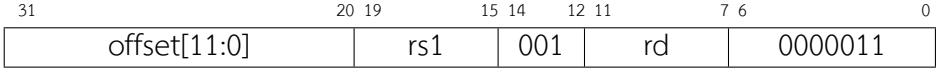
โหลดดับเบิลเวิร์ด (Load Doubleword). รูปแบบ-I, RV64I เท่านั้น.
โหลด 8 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + (*offset*) *ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*].
คำสั่งบีบอัด: **c.ldsp** rd, offset; **c.ld** rd, offset(rs1)



---

lh rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

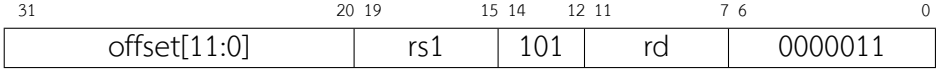
โหลดฮาล์ฟเวิร์ด (Load Halfword). รูปแบบ-I, RV32I และ RV64I.
โหลด 2 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*]+ (*offset*) *ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*] โดยทำการขยายเครื่องหมายให้กับผลลัพท์



---

lhu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][15:0]`

โหลดฮาล์ฟเวิร์ด, ไม่มีเครื่องหมาย (Load Halfword, Unsigned). รูปแบบ-I, RV32I และ RV64I.
โหลด 2 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*]+ (*offset*) *ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*] โดยทำการขยายศูนย์ให้กับผลลัพท์



---

li rd, immediate `x[rd] = immediate`

โหลดค่าคงที่ (Load Immediate). คำสั่งเทียม, RV32I และ RV64I.
โหลดค่าคงที่ลงใน x[*rd*] โดยใช้คำสั่งให้น้อยที่สุดเท่าที่จะเป็นไปได้ สำหรับ RV32I จะขยายเป็น **lui** และ/หรือ **addi**; สำหรับ RV64I จะยาวถึง **lui, addi, slli, addi, slli, addi, slli, addi**

---

lla rd, symbol `x[rd] = &symbol`

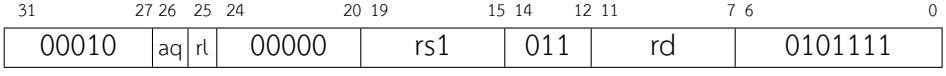
โหลดตที่อยู่ภายใน (Load Local Address). คำสั่งเทียม, RV32I และ RV64I.
โหลดที่อยู่ของสัญลักษณ์ *(symbol)* ลงใน x[*rd*] ขยายเป็น **auipc** rd, offsetHi จากนั้น insnaddird, rd, offsetLo

---

lr.d rd, (rs1) `x[rd] = LoadReserved64(M[x[rs1]])`

การดำเนินการหน่วยความจำแบบอะตอมิก: โหลดแบบจองหน่วยความจำดับเบิลเวิร์ด (Load-Reserved Doubleword). รูปแบบ-R, RV64A เท่านั้น.

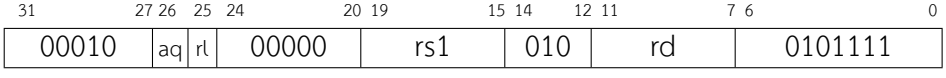
โหลดหน่วยความจำ 8 ไบต์ จากหน่วยความจำที่ตำแหน่งที่อยู่ ในรีจิสเตอร์ x[*rs1*] เขียนลงใน x[*rd*] และบันทึกการจองบนที่อยู่หน่วยความจำดับเบิลเวิร์ดนั้น



lr.w rd, (rs1) `x[rd] = LoadReserved32(M[x[rs1]])`

การดำเนินการหน่วยความจำแบบอะตอมิก: โหลดแบบจองในหน่วยความจำเวิร์ด (Load-Reserved Word). รูปแบบ-R, RV32A และ RV64A.

โหลดหน่วยความจำ 4 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ ในรีจิสเตอร์ x[*rs1*] เขียนลงใน x[*rd*] โดยขยายเครื่องหมายให้กับผลลัพท์ และบันทึกการจองบนหน่วยความจำเวิร์ดนั้น

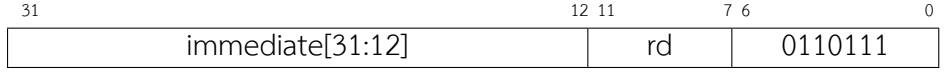


lui rd, immediate `x[rd] = sext(immediate[31:12] << 12)`

โหลดบิตบนค่าคงที่ทันที (Load Upper Immediate). รูปแบบ-U, RV32I และ RV64I.

เขียน*ค่าคงที่* ขนาด 20 บิตที่ขยายเครื่องหมาย และเลื่อนซ้าย 12 บิต ลงใน x[*rd*] โดยทำให้ 12 บิตล่างเป็นศูนย์

คำสั่งบีบอัด: **c.lui** rd, imm

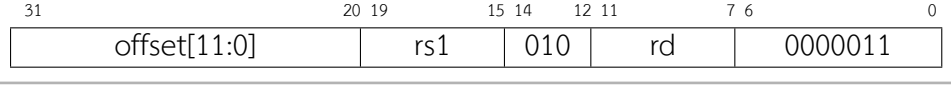


lw rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

โหลดเวิร์ด (Load Word). รูปแบบ-I, RV32I และ RV64I.

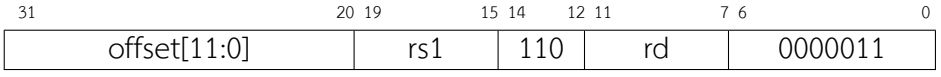
โหลด 4 ไบต์จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] (*offset*)*ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*] สำหรับ RV64I ผลลัพธ์จะถูกขยายเครื่องหมาย

คำสั่งบีบอัด: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)



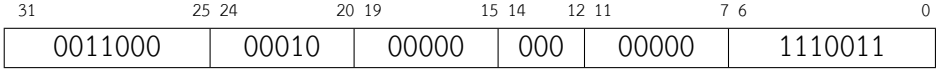
lwu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][31:0]`

โหลดเวิร์ด, ไม่มีเครื่องหมาย (Load Word, Unsigned). รูปแบบ-I, RV64I เท่านั้น.
โหลด 4 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ x[*rs1*]+ (*offset*) *ที่ขยายเครื่องหมาย* และเขียนลงใน x[*rd*] โดยทำการขยายศูนย์ให้กับผลลัพท์



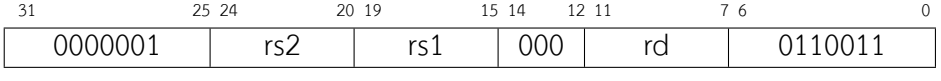
mret `ExceptionReturn(Machine)`

การคืนค่าจากข้อยกเว้นในโหมดเครื่อง (Machine-mode Exception Return). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.
การคืนค่าจากตัวจัดการข้อยกเว้นในโหมดเครื่อง ตั้งค่า *pc* ให้มีค่าเท่ากับ CSRs[mepc], ตั้งค่าโหมดสิทธิพิเศษให้เป็นค่าใน CSRs[mstatus].MPP, ตั้งค่า CSRs[mstatus].MIE ให้เป็นค่าใน CSRs[mstatus].MPIE และตั้งค่า CSRs[mstatus].MPIE ให้เป็น 1; และหากรองรับโหมดผู้ใช้ ให้ตั้งค่า CSRs[mstatus].MPP เป็น 0



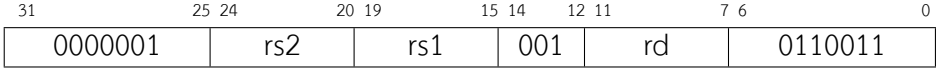
mul rd, rs1, rs2 `x[rd] = x[rs1] × x[rs2]`

การคูณ (Multiply). รูปแบบ-R, RV32M และ RV64M.
คูณ x[*rs1*] ด้วย x[*rs2*] และเขียนผลคูณลงใน x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ



mulh rd, rs1, rs2 $x[rd] = (x[rs1] \ {}_s\times_s \ x[rs2]) \gg_s XLEN$

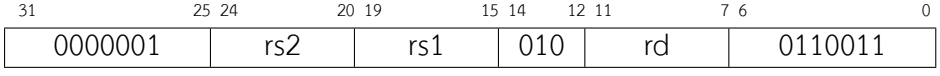
คูณบิตบน (Multiply High). รูปแบบ-R, RV32M และ RV64M.
คูณ x[*rs1*] กับ x[*rs2*] โดยถือว่าค่าเป็นตัวเลขแบบส่วนเติมเต็มสองและเขียนบิตครึ่งบนของผลคูณลงใน x[*rd*]



mulhsu rd, rs1, rs2 `x[rd] = (x[rs1] ` $_s\times_u$ `x[rs2]) >>` $_s$ `XLEN`

คูณบิตสูงแบบมีเครื่องหมาย-ไม่มีเครื่องหมาย (Multiply High Signed-Unsigned). รูปแบบ-R, RV32M และ RV64M.

คูณ x[*rs1*] กับ x[*rs2*] โดยถือว่า x[rs1] เป็นตัวเลขแบบส่วนเติมเต็มสองและ x[rs2] เป็นตัวเลขไม่มีเครื่องหมาย และเขียนบิตครึ่งบนของผลคูณ ลงใน x[*rd*]

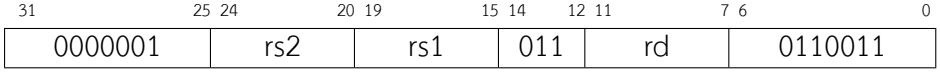


---

mulhu rd, rs1, rs2 `x[rd] = (x[rs1] ` $_u\times_u$ `x[rs2]) >>` $_u$ `XLEN`

คูณบิตสูงแบบไม่มีเครื่องหมาย (Multiply High Unsigned). รูปแบบ-R, RV32M และ RV64M.

คูณ x[*rs1*] กับ x[*rs2*] โดยถือว่าค่าเป็นตัวเลขที่ไม่มีเครื่องหมาย และเขียนบิตครึ่งบนของผลคูณลงใน x[*rd*]

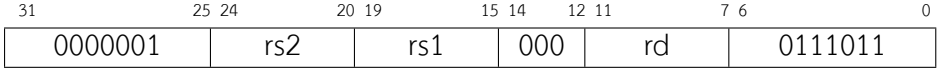


---

mulw rd, rs1, rs2 `x[rd] = sext((x[rs1] × x[rs2])[31:0])`

คูณเวิร์ด (Multiply Word). รูปแบบ-R, RV64M เท่านั้น.

คูณ x[*rs1*] กับ x[*rs2*] ตัดทอนผลคูณเป็น 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมายไปยัง x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ



---

mv rd, rs1 `x[rd] = x[rs1]`

ย้าย (Move). คำสั่งเทียม, RV32I และ RV64I.

คัดลอกข้อมูลจากรีจิสเตอร์ x[*rs1*] ไปยัง x[*rd*] ขยายเป็น **addi** rd, rs1, 0

---

neg rd, rs2 `x[rd] = -x[rs2]`

กลับค่าเป็นลบ (Negate). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่าส่วนเติมเต็มสองของ x[*rs2*] ลงใน x[*rd*] ขยายเป็น **sub** rd, x0, rs2

---

negw rd, rs2 `x[rd] = sext((-x[rs2])[31:0])`

กลับค่าเป็นลบแบบเวิร์ด (Negate Word). คำสั่งเทียม, RV64I เท่านั้น.

คำนวณแบบส่วนเติมเต็มสองของ x[*rs2*] ตัดทอนผลลัพธ์เป็น 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมายลงใน x[*rd*] ขยายเป็น **subw** rd, x0, rs2

---

nop

`Nothing`

ไม่ดำเนินการ (No operation). คำสั่งเทียม, RV32I และ RV64I.
เลื่อน *pc* ไปยังคำสั่งถัดไป โดยขยายเป็น **addi** x0, x0, 0

---

not rd, rs1

`x[rd] = ~x[rs1]`

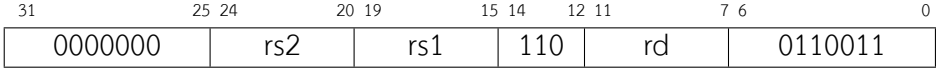
กลับค่าบิต (NOT). คำสั่งเทียม, RV32I และ RV64I.
เขียนผลลัพธ์แบบส่วนเติมเต็มหนึ่งของ x[*rs1*] ลงใน x[*rd*] ขยายเป็น **xori** rd, rs1, -1

---

or rd, rs1, rs2

`x[rd] = x[rs1] | x[rs2]`

ดำเนินการระดับบิต-OR (OR). รูปแบบ-R, RV32I และ RV64I.
คำนวณค่าระดับบิตรวม-OR ของรีจิสเตอร์ x[*rs1*] และ x[*rs2*] และเขียนผลลัพธ์ลงใน x[*rd*]
คำสั่งบีบอัด: **c.or** rd, rs2

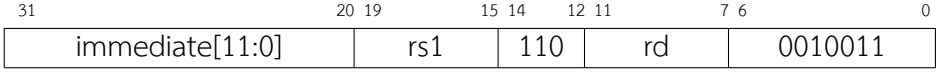


---

ori rd, rs1, immediate

`x[rd] = x[rs1] | sext(immediate)`

ดำเนินการระดับบิต-OR ค่าคงที่ (OR Immediate). รูปแบบ-I, RV32I และ RV64I.
คำนวณค่าระดับบิตรวม-OR ระหว่าง *ค่าคงที่*ที่ขยายเครื่องหมาย และรีจิสเตอร์ x[*rs1*]
และเขียนผลลัพธ์ลงใน x[*rd*]



---

rdcycle rd

`x[rd] = CSRs[cycle]`

อ่านตัวนับรอบ (Read Cycle Counter). คำสั่งเทียม, RV32I และ RV64I.
เขียนจำนวนรอบที่ผ่านไปลงใน x[*rd*] โดยขยายเป็น **csrrs** rd, cycle, x0

---

rdcycleh rd

`x[rd] = CSRs[cycleh]`

อ่านบิตบนของตัวนับรอบ (Read Cycle Counter High). คำสั่งเทียม, RV32I เท่านั้น.
เขียนบิตบนของจำนวนรอบที่ผ่านไป ซึ่งถูกเลื่อนขวา 32 บิต ลงใน x[*rd*] ขยายเป็น **csrrs** rd, cycleh, x0

---

rdinstret rd

`x[rd] = CSRs[instret]`

อ่านตัวนับคำสั่งที่สิ้นสุดการทำงาน (Read Instructions-Retired Counter). คำสั่งเทียม, RV32I และ RV64I.

เขียนจำนวนคำสั่งที่สิ้นสุดการทำงาน ลงใน [*rd*] โดยขยายเป็น **csrrs** rd, instret, x0

---

rdinstreth rd

`x[rd] = CSRs[instreth]`

อ่านตัวนับคำสั่งที่สิ้นสุดการทำงานบิตบน (Read Instructions-Retired Counter High). คำสั่งเทียม, RV32I เท่านั้น.

เขียนจำนวนคำสั่งที่สิ้นสุดการทำงาน ซึ่งถูกเลื่อนขวา 32 บิต ลงใน x[*rd*] ขยายเป็น **csrrs** rd, instreth, x0

---

rdtime rd

`x[rd] = CSRs[time]`

อ่านเวลา (Read Time). คำสั่งเทียม, RV32I และ RV64I.

เขียนเวลาปัจจุบัน ลงใน x[*rd*] ความถี่ของตัวจับเวลาขึ้นอยู่กับแพลตฟอร์ม โดยขยายเป็น **csrrs** rd, time, x0

---

rdtimeh rd

`x[rd] = CSRs[timeh]`

อ่านเวลาบิตสูง (Read Time High). คำสั่งเทียม, RV32I เท่านั้น.

เขียนเวลาปัจจุบัน ซึ่งถูกเลื่อนขวา 32 บิต ลงใน x[*rd*] ความถี่ของตัวจับเวลาขึ้นอยู่กับแพลตฟอร์ม โดยขยายเป็น **csrrs** rd, timeh, x0

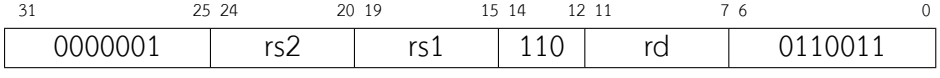
---

rem rd, rs1, rs2

`x[rd] = x[rs1]` $\%_s$ `x[rs2]`

เศษ (Remainder). รูปแบบ-R, RV32M และ RV64M.

หาร x[*rs1*] ด้วย x[*rs2*] โดยปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบส่วนเติมเต็มสอง) และเขียนเศษไปยัง x[*rd*]



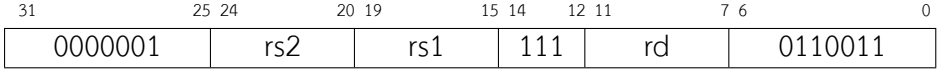
---

remu rd, rs1, rs2

`x[rd] = x[rs1]` $\%_u$ `x[rs2]`

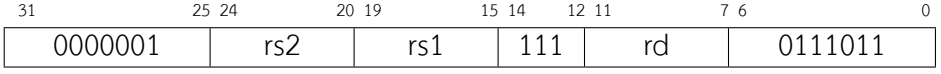
เศษ, ไม่มีเครื่องหมาย (Remainder, Unsigned). รูปแบบ-R, RV32M และ RV64M.

หาร x[*rs1*] ด้วย x[*rs2*] โดยปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียนเศษไปยัง x[*rd*]



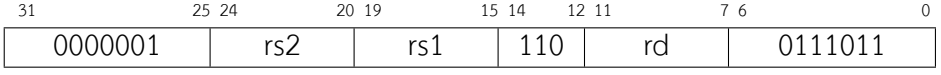
remuw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %u x[rs2][31:0])`

เศษเวิร์ด, ไม่มีเครื่องหมาย(Remainder Word, Unsigned). รูปแบบ-R, RV64M เท่านั้น.
หาร 32 บิตล่างของ x[*rs1*] ด้วย 32 บิตล่าวของ x[*rs2*] โดยปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียนเศษ 32 บิตที่ขยายเครื่องหมาย ไปยัง x[*rd*]



remw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %s x[rs2][31:0])`

เศษเวิร์ด (Remainder Word). รูปแบบ-R, RV64M เท่านั้น.
หาร 32 บิตล่างของ x[*rs1*] ด้วยค่า 32 บิตล่างของ x[*rs2*] โดยปัดเศษเข้าหาศูนย์ ถือว่าค่าเป็นตัวเลขแบบส่วนเติมเต็มสองและเขียนเศษ 32 บิตที่ขยายเครื่องหมาย ไปยัง x[*rd*]

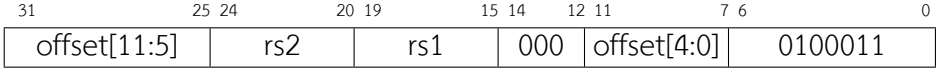


ret `pc = x[1]`

การกลับจากฟังก์ชัน (Return). คำสั่งเทียม, RV32I และ RV64I.
คืนค่าจากรูทีนย่อย ขยายเป็น **jalr** x0, 0(x1)

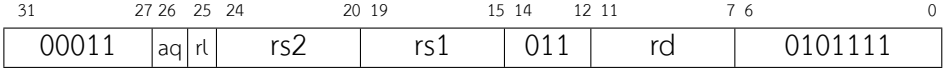
sb rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][7:0]`

จัดเก็บไบต์ (Store Byte). รูปแบบ-S, RV32I และ RV64I.
เก็บไบต์ที่มีนัยสำคัญน้อยที่สุด ในรีจิสเตอร์ x[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + *sign-extend*(*offset*)



sc.d rd, rs2, (rs1) `x[rd] = StoreConditional64(M[x[rs1]], x[rs2])`

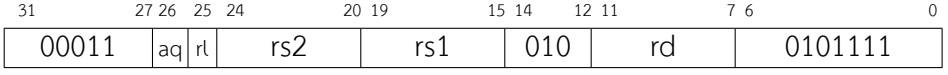
การดำเนินการหน่วยความจำแบบอะตอมิก: จัดเก็บแบบมีเงื่อนไขดับเบิลเวิร์ด (Store-Conditional Doubleword). รูปแบบ-R, RV64A เท่านั้น.
จัดเก็บ 8 ไบต์ ในรีจิสเตอร์ x[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] โดยมีเงื่อนไขว่าต้องมีการจองการโหลด บนที่อยู่นั้นอยู่ก่อน เขียนค่า 0 ลงใน x[*rd*] หากการจัดเก็บสำเร็จ หรือเขียนรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ หากไม่สำเร็จ



sc.w rd, rs2, (rs1) `x[rd] = StoreConditional32(M[x[rs1]], x[rs2])`

การดำเนินการหน่วยความจำแบบอะตอมิก: จัดเก็บเวิร์ดแบบมีเงื่อนไข (Store-Conditional Word). รูปแบบ-R, RV32A และ RV64A.

จัดเก็บ 4 ไบต์ ในรีจิสเตอร์ x[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] โดยมีเงื่อนไขว่าต้องมีการจองการโหลด บนที่อยู่นั้นอยู่ก่อน เขียนค่า 0 ลงใน x[*rd*] หากการจัดเก็บสำเร็จ หรือเขียนรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ หากไม่สำเร็จ



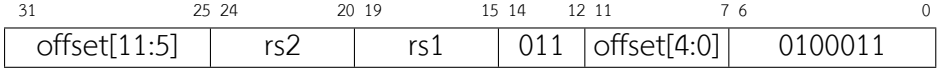
---

sd rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][63:0]`

จัดเก็บดับเบิลเวิร์ด (Store Doubleword). รูปแบบ-S, RV64I เท่านั้น.

เก็บ 8 ไบต์ในรีจิสเตอร์ x[*rs2*] ไปยังหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + *sign-extend*(*offset*) โดย*ขยายเครื่องหมาย*(*offset*)

คำสั่งบีบอัด: **c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)



---

seqz rd, rs1 `x[rd] = (x[rs1] == 0)`

ตั้งค่าเมื่อเท่ากับศูนย์(Set if Equal to Zero). คำสั่งเทียม, RV32I and RV64I.

เขียนค่า 1 ลงใน x[*rd*] ถ้า x[*rs1*] เท่ากับ 0 หรือเขียนค่า 0 ถ้าไม่ใช่ โดยขยายเป็น **sltiu** rd, rs1, 1

---

sext.w rd, rs1 `x[rd] = sext(x[rs1][31:0])`

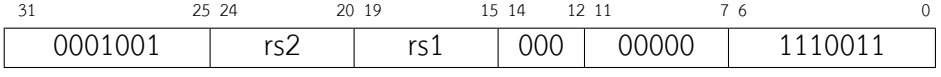
ขยายเครื่องหมายเวิร์ด (Sign-extend Word). คำสั่งเทียม, RV64I only.

อ่าน 32 บิตล่างของ x[*rs1*] ขยายเครื่องหมาย และเขียนผลลัพธ์ลงใน x[*rd*] โดยขยายเป็น **addiw** rd, rs1, 0

---

sfence.vma rs1,rs2 `Fence(Store, AddressTranslation)`

การควบคุมลำดับการทำงานของหน่วยความจำด้วย Fence (Fence Virtual Memory). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.
จัดการลำดับการเก็บข้อมูลไปยังตารางหน้าดำเนินการเสร็จแล้วแปลงที่อยู่ใหม่ โดยดึงข้อมูลจากตารางหน้าที่อัพเดตล่าสุด เมื่อ *rs2*=0 จะล้างทุก address space หากไม่ใช่ 0 จะล้างเฉพาะ address space ที่ระบุโดย x[*rs2*] เมื่อ *rs1*=0 จะล้างทุก virtual address ใน address space ที่เลือก หากไม่ใช่ 0 จะล้างเฉพาะหน้า (page) ที่มีที่อยู่หน่วยความจำเสมือน x[*rs1*] ใน address space ที่เลือก



---

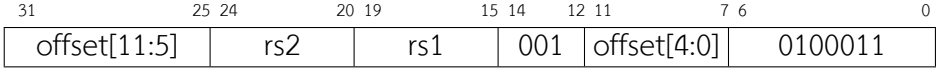
sgtz rd, rs2 `x[rd] = (x[rs2] >s 0)`

การตั้งค่าถ้ามากกว่าศูนย์ (Set if Greater Than to Zero). คำสั่งเทียม, RV32I และ RV64I.
เขียน 1 ไปยัง x[*rd*] ถ้า x[*rs2*] มีค่ามากกว่า 0 หรือ เขียน 0 ถ้าไม่มากกว่า ขยายเป็น **slt** rd, x0, rs2

---

sh rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][15:0]`

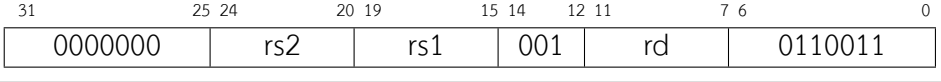
การจัดเก็บแบบฮาล์ฟเวิร์ด (Store Halfword). รูปแบบ-S, RV32I และ RV64I.
เก็บไบต์ที่มีนัยสำคัญน้อยที่สุดสองไบต์จากรีจิสเตอร์ x[*rs2*] ลงในหน่วยความจำตำแหน่งที่อยู่ x[*rs1*]+ *sign-extend*(*offset*)



---

sll rd, rs1, rs2 `x[rd] = x[rs1] << x[rs2]`

การเลื่อนบิตซ้ายเชิงตรรกะ (Shift Left Logical). รูปแบบ-R, RV32I และ RV64I.
เลื่อนรีจิสเตอร์ x[*rs1*] ไปซ้ายตามจำนวนตำแหน่งบิตที่กำหนดใน x[*rs2*] บิตที่ว่างจะถูกเติมด้วยศูนย์ และผลลัพธ์จะถูกเขียนลงใน x[*rd*] โดยใช้ห้าบิตที่มีนัยสำคัญน้อยที่สุดของ x[*rs2*] (หรือ หกบิตสำหรับ RV64I) เป็นจำนวนการเลื่อนบิต (เฉพาะ 5 บิตล่างของ rs2) บิตบนจะถูกละเว้น

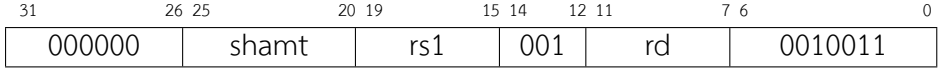


---

slli rd, rs1, shamt `x[rd] = x[rs1] << shamt`

การเลื่อนบิตขวาเชิงตรรกะทันที (Shift Left Logical Immediate). รูปแบบ-I, RV32I และ RV64I.

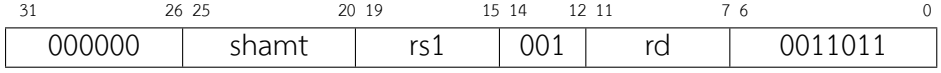
เลื่อนรีจิสเตอร์ x[*rs1*] ไปซ้ายตามจำนวนตำแหน่งบิตที่กำหนดใน *shamt* บิตที่ว่างจะถูกเติมด้วยศูนย์ และผลลัพธ์จะถูกเขียนไปที่ x[*rd*] สำหรับ RV32I คำสั่งนี้จะถูกต้องก็ต่อเมื่อ *shamt*[5]=0 emphคำสั่งบีบอัด: **c.slli** rd, shamt



slliw rd, rs1, shamt `x[rd] = sext((x[rs1] << shamt)[31:0])`

การเลื่อนบิตซ้ายเชิงตรรกะทันทีแบบเวิร์ด (Shift Left Logical Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

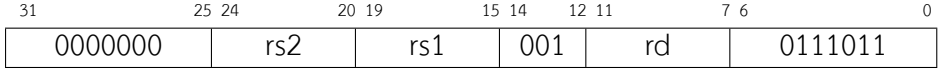
เลื่อนรีจิสเตอร์ x[*rs1*] ไปซ้ายตามตำแหน่งบิต *shamt* บิตที่ว่างจะถูกเติมด้วยศูนย์ ผลลัพธ์จะถูกตัดให้เหลือ 32 บิต และผลลัพธ์ 32 บิตที่ขยายเครื่องหมายจะถูกเขียนไปยัง x[*rd*] คำสั่งจะถูกต้องก็ต่อเมื่อ *shamt*[5]=0



sllw rd, rs1, rs2 `x[rd] = sext((x[rs1] << x[rs2][4:0])[31:0])`

การเลื่อนบิตซ้ายเชิงตรรกะแบบเวิร์ด (Shift Left Logical Word). รูปแบบ-R, RV64I เท่านั้น.

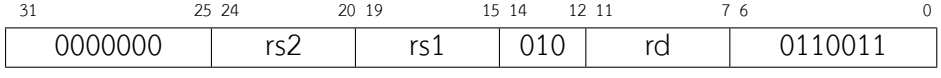
เลื่อน 32 บิตล่างของ x[*rs1*] ไปซ้ายตามตำแหน่งบิต x[*rs2*] บิตที่ว่างจะถูกเติมด้วยศูนย์ และผลลัพธ์ 32 บิตที่ขยายเครื่องหมายจะ ถูกเขียนไปยัง x[*rd*] โดยบิตที่มีนัยสำคัญน้อยที่สุดจำนวนห้าบิตของ x[*rs2*] จะเป็นจำนวนบิตที่เลื่อน บิตบนจะถูกละเว้น



slt rd, rs1, rs2 `x[rd] = x[rs1]` $<_s$ `x[rs2]`

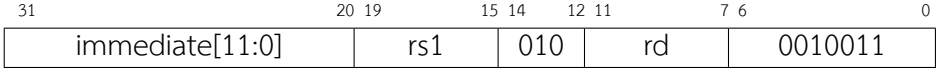
ตั้งค่าถ้าน้อยกว่า (Set if Less Than). รูปแบบ-R, RV32I และ RV64I.

เปรียบเทียบ x[*rs1*] กับค่า x[*rs2*] แบบส่วนเติมเต็มสองและเขียน 1 ไปที่ x[*rd*] ถ้า x[*rs1*] น้อยกว่า หรือเขียน 0 ถ้าไม่น้อยกว่า



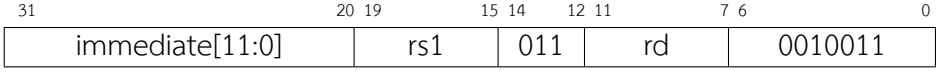
slti rd, rs1, immediate `x[rd] = x[rs1] <s sext(immediate)`

ตั้งค่าถ้าน้อยกว่าทันที (Set if Less Than Immediate). รูปแบบ-I, RV32I และ RV64I. เปรียบเทียบ x[*rs1*] กับค่า *ค่าคงที่*ที่ถูกขยายเครื่องหมาย เป็นตัวเลขแบบส่วนเติมเต็มสอง และเขียน 1 ไปที่ x[*rd*] ถ้า x[*rs1*] มีค่าน้อยกว่า หรือเขียน 0 ถ้าไม่น้อยกว่า



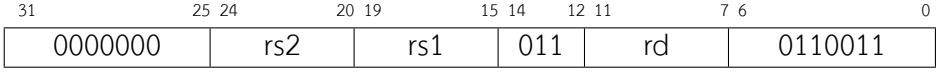
sltiu rd, rs1, immediate `x[rd] = x[rs1] <u sext(immediate)`

ตั้งค่าถ้าน้อยกว่าแบบทันทีไม่มีเครื่องหมาย (Set if Less Than Immediate, Unsigned). รูปแบบ-I, RV32I และ RV64I.
เปรียบเทียบ x[*rs1*] กับค่า*ค่าคงที่* ที่ขยายเครื่องหมาย เป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียนค่า 1 ลงใน x[*rd*] ถ้า x[*rs1*] น้อยกว่า หรือเขียน 0 ถ้าไม่น้อยกว่า



sltu rd, rs1, rs2 `x[rd] = x[rs1] <u x[rs2]`

ตั้งค่าถ้าน้อยกว่าแบบไม่มีเครื่องหมาย (Set if Less Than, Unsigned). รูปแบบ-R, RV32I และ RV64I.
เปรียบเทียบ x[*rs1*] กับ x[*rs2*] เป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียน 1 ลงใน x[*rd*] ถ้า x[*rs1*] น้อยกว่า ถ้าไม่น้อยกว่าให้เขียน 0



sltz rd, rs1 `x[rd] = (x[rs1] <s 0)`

ตั้งค่าถ้าน้อยกว่าศูนย์ (Set if Less Than to Zero). คำสั่งเทียม, RV32I และ RV64I. เขียน 1 ลงใน x[*rd*] ถ้า x[*rs1*] น้อยกว่าศูนย์ หรือเขียน 0 ถ้าไม่น้อยกว่า ขยายเป็น **slt** rd, rs1, x0

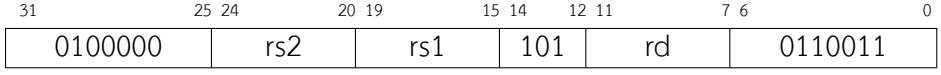
---

snez rd, rs2 `x[rd] = (x[rs2] ≠ 0)`

ตั้งค่าถ้าไม่เท่ากับศูนย์ (Set if Not Equal to Zero). คำสั่งเทียม, RV32I และ RV64I. เขียนค่า 0 ลงใน x[*rd*] ถ้า x[*rs2*] เท่ากับ 0 หรือเขียนค่า 1 ถ้าไม่เท่ากับ 0 ขยายเป็น **sltu** rd, x0, rs2

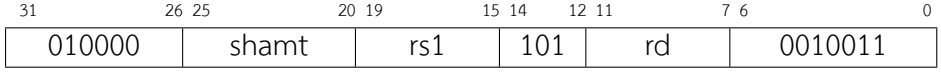
sra rd, rs1, rs2 `x[rd] = x[rs1] >>s x[rs2]`

เลื่อนบิตไปขวาเชิงคณิตศาสตร์ (Shift Right Arithmetic). รูปแบบ-R, RV32I และ RV64I.
เลื่อนรีจิสเตอร์ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน x[*rs2*] โดยบิตที่ว่างลงจะถูกเติมด้วยสำเนาของบิตที่มีนัยสำคัญสูงสุดของ x[*rs1*] และผลลัพธ์จะถูกเขียนลงใน x[*rd*] จำนวนบิตที่ใช้ในการเลื่อนจะมาจากบิตที่มีนัยสำคัญน้อยที่สุดห้าบิตของ x[*rs2*] (หรือ 6 บิตสำหรับ RV64I) ส่วนบิตบนจะถูกละเว้น



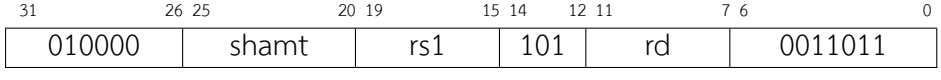
srai rd, rs1, shamt `x[rd] = x[rs1] >>s shamt`

เลื่อนบิตไปขวาเชิงคณิตศาสตร์ทันที (Shift Right Arithmetic Immediate). รูปแบบ-I, RV32I และ RV64I.
เลื่อนรีจิสเตอร์ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน *shamt* บิตที่ว่างจะถูกเติมด้วยสำเนาของบิตที่มีนัยสำคัญสูงสุดของ x[*rs1*] และผลลัพธ์จะถูกเขียนลงใน x[*rd*] สำหรับ RV32I คำสั่งนี้จะถูกต้องตามกฎก็ต่อเมื่อ *shamt*[5]=0.
คำสั่งบีบอัด: **c.srai** rd, shamt



sraiw rd, rs1, shamt `x[rd] = sext(x[rs1][31:0] >>s shamt)`

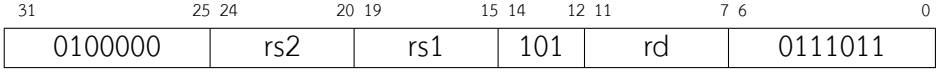
เลื่อนบิตไปขวาเชิงคณิตศาสตร์แบบเวิร์ดทันที (Shift Right Arithmetic Word Immediate). รูปแบบ-I, RV64I เท่านั้น.
เลื่อน 32 บิตล่างของรีจิสเตอร์ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน *shamt* บิตที่ว่างจะเติมด้วยสำเนาของ x[*rs1*][31] และผลลัพธ์ 32 บิตที่ถูกขยายเครื่องหมาย จะถูกเขียนลงใน x[*rd*] คำสั่งนี้จะถูกต้องตามกฎก็ต่อเมื่อ *shamt*[5]=0



sraw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] >>s x[rs2][4:0])`

เลื่อนบิตไปขวาเชิงคณิตศาสตร์แบบเวิร์ด (Shift Right Arithmetic Word). รูปแบบ-R, RV64I เท่านั้น.

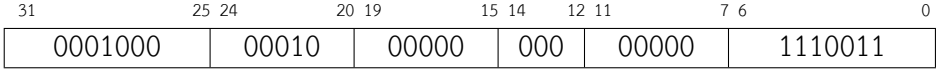
เลื่อนบิต 32 บิตล่างของ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน x[*rs2*] โดยบิตที่ว่างลงจะถูกเติมด้วย x[*rs1*][31] และผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย จะถูกเขียนไปยัง x[*rd*] จำนวนบิตที่ใช้ในการเลื่อนจะมาจากบิตที่มีนัยสำคัญน้อยที่สุดห้าบิตของ x[*rs2*] ส่วนบิตบนจะถูกละเว้น



sret `ExceptionReturn(Supervisor)`

การคืนค่าจากข้อยกเว้นในโหมดผู้ดูแลระบบ (Supervisor-mode Exception Return). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.

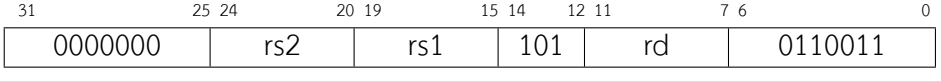
คืนค่าจากตัวจัดการข้อยกเว้นโหมดซุปเปอร์ไวเซอร์ โดยกำหนดค่า *pc* เป็น CSRs[sepc] โหมดสิทธิพิเศษเป็น CSRs[sstatus].SPP, CSRs[sstatus].SIE เป็น CSRs[sstatus].SPIE, CSRs[sstatus].SPIE เป็น 1 และ CSRs[สถานะ].SPP เป็น 0



srl rd, rs1, rs2 `x[rd] = x[rs1] >>u x[rs2]`

เลื่อนบิตไปขวาเชิงตรรกะ (Shift Right Logical). รูปแบบ-R, RV32I และ RV64I.

เลื่อนรีจิสเตอร์ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน x[*rs2*] โดยบิตที่ว่างจะถูกแทนที่ด้วยศูนย์ และผลลัพธ์จะถูกเขียนไปยัง x[*rd*] จำนวนบิตที่ใช้ในการเลื่อนจะมาจากบิตที่มีนัยสำคัญน้อยที่สุดห้าบิตของ x[*rs2*] (หรือ 6 บิตสำหรับ RV64I) บิตบนจะถูกละเว้น

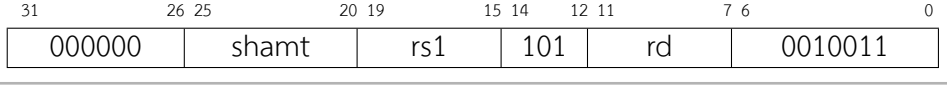


srli rd, rs1, shamt `x[rd] = x[rs1] >>u shamt`

เลื่อนบิตไปขวาเชิงตรรกะแบบทันที (Shift Right Logical Immediate). รูปแบบ-I, RV32I และ RV64I.

เลื่อนรีจิสเตอร์ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน *shamt* โดยบิตที่ว่างลงจะถูกเติมด้วยศูนย์ และผลลัพธ์จะถูกเขียนลงใน x[*rd*] สำหรับ RV32I คำสั่งนี้จะถูกต้องก็ต่อเมื่อ *shamt*[5]=0.

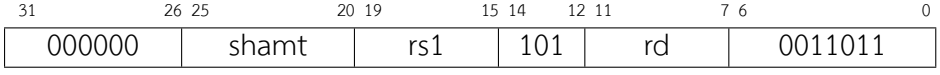
คำสั่งบีบอัด: **c.srli** rd, shamt



srliw rd, rs1, shamt `x[rd] = sext(x[rs1][31:0] >>u shamt)`

เลื่อนบิตไปขวาเชิงตรรกะแบบเวิร์ดทันที (Shift Right Logical Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

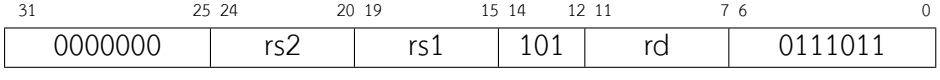
เลื่อนบิต 32 บิตล่างของ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน *shamt* โดยบิตที่ว่างจะถูกเติมด้วยศูนย์ และผลลัพธ์ 32 บิตที่ถูกขยายเครื่องหมายจะถูกเขียนลงใน x[*rd*] คำสั่งจะถูกต้องก็ต่อเมื่อ *shamt*[5]=0



srlw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] >>u x[rs2][4:0])`

เลื่อนบิตไปขวาเชิงตรรกะแบบเวิร์ด (Shift Right Logical Word). รูปแบบ-R, RV64I เท่านั้น.

เลื่อนบิต 32 บิตล่างของ x[*rs1*] ไปขวาตามจำนวนตำแหน่งบิตที่ระบุใน x[*rs2*] โดยบิตที่ว่างจะถูกเติมด้วยศูนย์ และผลลัพธ์ 32 บิตที่ถูกขยายเครื่องหมายจะถูกเขียนลงใน x[*rd*] จำนวนบิตที่ใช้ในการเลื่อนจะมาจากบิตที่มีนัยสำคัญน้อยที่สุดห้าบิตของ x[*rs2*] บิตบนจะถูกละเว้น

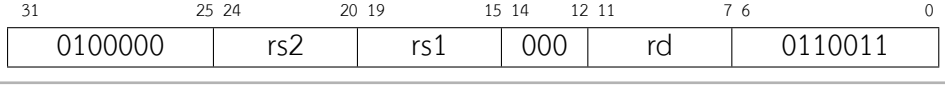


sub rd, rs1, rs2 `x[rd] = x[rs1] - x[rs2]`

การลบ (Subtract). รูปแบบ-R, RV32I และ RV64I.

ลบรีจิสเตอร์ x[*rs2*] จาก รีจิสเตอร์ x[*rs1*] และเขียนผลลัพธ์ไปยัง x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.sub** rd, rs2

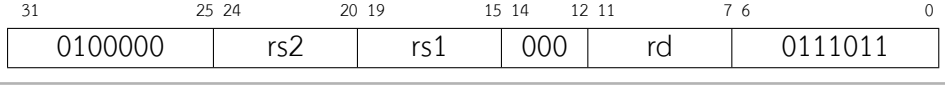


subw rd, rs1, rs2 `x[rd] = sext((x[rs1] - x[rs2])[31:0])`

ลบเวิร์ด (Subtract Word). รูปแบบ-R, RV64I เท่านั้น.

ลบ รีจิสเตอร์ x[*rs2*] จากรีจิสเตอร์ x[*rs1*] ตัดทอนผลลัพธ์เป็น 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมายไปที่ x[*rd*] ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.subw** rd, rs2

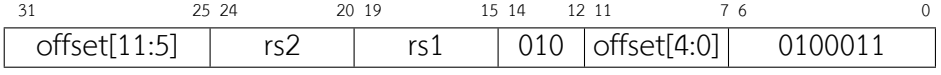


sw rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][31:0]`

จัดเก็บเวิร์ด (Store Word). รูปแบบ-S, RV32I และ RV64I.

เก็บไบต์ที่มีนัยสำคัญน้อยที่สุด 4 ไบต์ ในรีจิสเตอร์ x[*rs2*] ไปยังหน่วยความจำตำแหน่งที่อยู่ x[*rs1*] + *sign-extend*(*offset*).

คำสั่งบีบอัด: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)



31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011

---

tail symbol `pc = &symbol; clobber x[6]`

การเรียกฟังก์ชันท้ายสุด (Tail call). คำสั่งเทียม, RV32I และ RV64I.

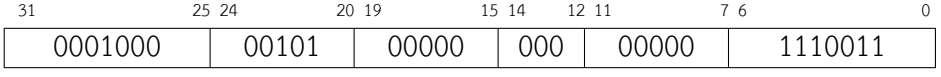
กำหนดค่า *pc* เป็น *symbol* โดยในกระบวนการนี้จะเขียนทับค่าใน x[6] ขยายเป็น **auipc** x6, offsetHi จากนั้นตามด้วย **jalr** x0, offsetLo(x6)

---

wfi `while (noInterruptsPending) idle`

รอการขัดจังหวะ (Wait for Interrupt). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.

ทำให้โปรเซสเซอร์อยู่ในสถานะหยุดทำงานเพื่อประหยัดพลังงาน หากไม่มีการขัดจังหวะที่รอดำเนินการถูกเปิดใช้งานอยู่



31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

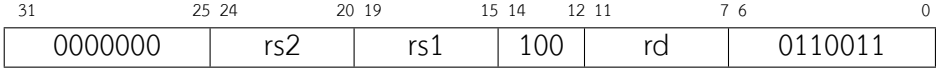
---

xor rd, rs1, rs2 `x[rd] = x[rs1] ^ x[rs2]`

เอกซ์คลูซีฟ-OR (Exclusive-OR). รูปแบบ-R, RV32I และ RV64I.

คำนวณค่าระดับบิต X-OR ของรีจิสเตอร์ x[*rs1*] และ x[*rs2*] และเขียนผลลัพธ์ลงใน x[*rd*]

คำสั่งบีบอัด: **c.xor** rd, rs2



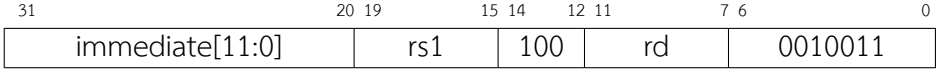
31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	100	rd	0110011

---

xori rd, rs1, immediate `x[rd] = x[rs1] ^ sext(immediate)`

เอกซ์คลูซีฟ-OR ค่าคงที่ (Exclusive-OR Immediate). รูปแบบ-I, RV32I และ RV64I.

คำนวณค่าระดับบิต X-OR ของ*ค่าคงที่*ขยายเครื่องหมาย และ x[*rs1*] และเขียนผลลัพธ์ลงใน x[*rd*]

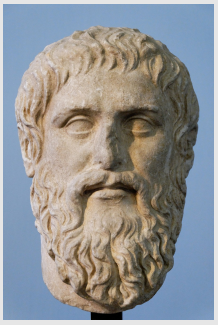


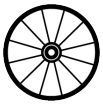
31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	100	rd	0010011

B การแปลงคำสั่ง RISC-V

ความงามของรูปแบบ ความกลมกลืน ความสง่างาม และจังหวะที่ดีขึ้นอยู่กับความเรียบง่าย
—เพลโต, *สาธารณรัฐ*

เพลโต (พ.ศ.971–891 ก่อนคริสต์ศักราช) เป็น นักปรัชญากรีกโบราณผู้วางรากฐานให้กับคณิตศาสตร์ ปรัชญา และวิทยาศาสตร์ตะวันตก





ความเรียบง่าย

B.1 บทนำ

ภาคผนวกนี้ประกอบด้วยตารางการแปลงคำสั่งและรูปแบบการเขียนโค้ดทั่วไปใน RV32I เป็นโค้ดเทียบเท่าใน ARM-32 และ x86-32 เป้าหมายของการเขียนภาคผนวกนี้คือเพื่อช่วยโปรแกรมเมอร์ที่ไม่คุ้นเคยกับ RISC-V แต่มีความรู้เกี่ยวกับ ARM-32 หรือ x86-32 ให้เรียนรู้ RISC-V และช่วยในการแปลง ISA เก่าเป็นโค้ด RISC-V พื้นฐาน ภาคผนวกนี้จบด้วยโค้ดภาษา C ที่สำรวจต้นไม้ทวิภาค (binary tree) พร้อมโค้ดแอสเซมบลีที่มีคำอธิบายสำหรับทั้งสาม ISA โดยเราได้จัดลำดับคำสั่งของทั้งสามการพัฒนาให้คล้ายกันมากที่สุดเพื่อให้เห็นความสอดคล้องกัน

คำสั่งถ่ายโอนข้อมูลในรูปที่ B.1 แสดงความคล้ายคลึงระหว่างการโหลด(loads) และการบันทึก (stores) ของ RV32I และ ARM-32 สำหรับโหมดการกำหนดที่อยู่ที่นิยมมากที่สุด เนื่องจาก x86 ISA มีลักษณะมุ่งเน้นหน่วยความจำ-รีจิสเตอร์ (memory-register orientation) แทนที่จะเป็นแบบโหลด-บันทึก (load-store orientation) อย่าง RV32I และ ARM-32 ดังนั้น x86 จึงถ่ายโอนข้อมูลโดยใช้คำสั่งย้าย (move)

นอกเหนือจากคำสั่งเลขคณิต คำสั่งตรรกะ และคำสั่งเลื่อนบิตมาตรฐานแล้ว รูปที่ B.2 แสดงวิธีการดำเนินการทั่วไปในแต่ละ ISA ตัวอย่างเช่น การทำให้เรจิสเตอร์เป็นศูนย์ใช้

Description	RV32I	ARM-32	x86-32
Load word	`lw t0, 4(t1)`	`ldr r0, [r1, #4]`	`mov eax, [edi+4]`
Load halfword	`lh t0, 4(t1)`	`ldrsh r0, [r1, #4]`	`movsx eax,WORD PTR[edi+4]`
Load halfword unsigned	`lhu t0, 4(t1)`	`ldrh r0, [r1, #4]`	`movzx eax,WORD PTR[edi+4]`
Load byte	`lb t0, 4(t1)`	`ldrsb r0, [r1, #4]`	`movsx eax,BYTE PTR[edi+4]`
Load byte unsigned	`lbu t0, 4(t1)`	`ldrb r0, [r1, #4]`	`movzx eax,BYTE PTR[edi+4]`
Store byte	`sb t0, 4(t1)`	`strb r0, [r1, #4]`	`mov [edi+4], al`
Store halfword	`sh t0, 4(t1)`	`strh r0, [r1, #4]`	`mov [edi+4], ax`
Store word	`sw t0, 4(t1)`	`str r0, [r1, #4]`	`mov [edi+4], eax`

รูปที่ B.1: คำสั่งการเข้าถึงหน่วยความจำ RV32I แปลงเป็น ARM-32 และ x86-32

Description	RV32I	ARM-32	x86-32
Zero register	`li t0, 0`	`mov r0, #0`	`xor eax, eax`
Move register	`mv t0, t1`	`mov r0, r1`	`mov eax, edi`
Complement register	`not t0, t1`	`mvn r0, r1`	`not eax, edi`
Negate register	`neg t0, t1`	`rsb r0, r1, #0`	`mov eax, edi` `neg eax`
Load large constant	`lui t0, 0xABCDE` `addi t0, t0, 0x123`	`movw r0, #0xE123` `movt r0, #0xABCD`	`mov eax, 0xABCDE123`
Move PC to register	`auipc t0, 0`	`ldr r0, [pc, #-8]`	`call 1f` `1: pop eax`
Add	`add t0, t1, t2`	`add r0, r1, r2`	`lea eax, [edi+esi]`
Add (imm.)	`addi t0, t0, 1`	`add r0, r0, #1`	`add eax, 1`
Subtract	`sub t0, t0, t1`	`sub r0, r0, r1`	`sub eax, edi`

Set reg. to (reg=0)	`sltiu t0, t1, 1`	`rsbs r0, r1, #1`
		`movcc r0, #0`
			`xor eax, eax`
			`test edx, edx`
			`sete al`

Set reg. to (reg≠0)	`sltu t0, x0, t1`	`adds r0, r1, #0`
		`movne r0, #1`
			`xor eax, eax`
			`test edx, edx`
			`setne al`

Bitwise-OR	`or t0, t0, t1`	`orr r0, r0, r1`	`or eax, edi`
Bitwise-AND	`and t0, t0, t1`	`and r0, r0, r1`	`and eax, edi`
Bitwise-XOR	`xor t0, t0, t1`	`eor r0, r0, r1`	`xor eax, edi`
Bitwise-OR (imm.)	`ori t0, t0, 1`	`orr r0, r0, #1`	`or eax, 1`
Bitwise-AND (imm.)	`andi t0, t0, 1`	`and r0, r0, #1`	`and eax, 1`
Bitwise-XOR (imm.)	`xori t0, t0, 1`	`eor r0, r0, #1`	`xor eax, 1`
Shift left	`sll t0, t0, t1`	`lsl r0, r0, r1`	`sal eax, cl`
Shift right logical	`srl t0, t0, t1`	`lsr r0, r0, r1`	`shr eax, cl`
Shift right arith.	`sra t0, t0, t1`	`asr r0, r0, r1`	`sar eax, cl`
Shift left (imm.)	`slli t0, t0, 1`	`lsl r0, r0, #1`	`sal eax, 1`
Shift right logical (imm.)	`srli t0, t0, 1`	`lsr r0, r0, #1`	`shr eax, 1`
Shift right arith. (imm.)	`srai t0, t0, 1`	`asr r0, r0, #1`	`sar eax, 1`

รูปที่ B.2: คำสั่งเลขคณิต RV32I แปลงเป็น ARM-32 และ x86-32 รูปแบบคำสั่งสองตัวถูกดำเนินการ (two-operand) ของ x86-32 มักต้องใช้คำสั่งมากกว่ารูปแบบสามตัวถูกดำเนินการ (three-operand) ของ ARM-32 และ RV32I

Description	RV32I	ARM-32	x86-32
Branch if =	`beq t0, t1, foo`	`cmp r0, r1` `beq foo`	`cmp eax, esi` `je foo`
Branch if ≠	`bne t0, t1, foo`	`cmp r0, r1` `bne foo`	`cmp eax, esi` `jne foo`
Branch if <	`blt t0, t1, foo`	`cmp r0, r1` `blt foo`	`cmp eax, esi` `jl foo`
Branch if $\geq_s$	`bge t0, t1, foo`	`cmp r0, r1` `bge foo`	`cmp eax, esi` `jge foo`
Branch if $<_u$	`bltu t0, t1, foo`	`cmp r0, r1` `bcc foo`	`cmp eax, esi` `jb foo`
Branch if $\geq_u$	`bgeu t0, t1, foo`	`cmp r0, r1` `bcs foo`	`cmp eax, esi` `jnb foo`
Branch if =0	`beqz t0, foo`	`cmp r0, #0` `beq foo`	`test eax, eax` `je foo`
Branch if ≠0	`bnez t0, foo`	`cmp r0, #0` `bne foo`	`test eax, eax` `jne foo`
Direct jump or tail call	`jal x0, foo`	`b foo`	`jmp foo`
Subroutine call	`jal ra, foo`	`bl foo`	`call foo`
Subroutine return	`jalr x0, 0(ra)`	`bx lr`	`ret`
Indirect call	`jalr ra, 0(t0)`	`blx r0`	`call eax`
Indirect jump or tail call	`jalr x0, 0(t0)`	`bx r0`	`jmp eax`

รูปที่ B.3: คำสั่งการควบคุม RV32I แปลงเป็น ARM-32 และ x86-32 คำสั่งเปรียบเทียบและกระโดด (compare-and-branch) ของ RV32I ใช้จำนวนคำสั่งเพียงครึ่งหนึ่งเมื่อเทียบกับการกระโดดตามรหัสเงื่อนไข (condition code based branches) ของ ARM-32 และ x86-32

คำสั่งเทียม `li` ใน RV32I คำสั่งคัดลอกค่าคงที่ (move immediate) ใน ARM-32 และการดำเนินการ exclusive-OR รีจิสเตอร์กับตัวเองใน x86-32 ข้อจำกัดสองตัวถูกดำเนินการของคำสั่ง x86-32 ทำให้ต้องใช้คำสั่งมากขึ้นในบางกรณี แม้ว่ารูปแบบคำสั่งความยาวแปรผัน (variable length instruction format) จะช่วยให้โหลดค่าคงที่ขนาดใหญ่ได้ในคำสั่งเดียว คำสั่งบวก ลบ ตรรกะ และเลื่อนตำแหน่งแบบทั่วไป—ซึ่งเป็นคำสั่งที่ใช้มากที่สุด—มีการแมปแบบหนึ่งต่อหนึ่งระหว่าง ISA

รูปที่ B.3 รายการคำสั่งกระโดดแบบมีเงื่อนไขและไม่มีเงื่อนไข รวมถึงคำสั่งเรียก (call instructions) แสดงให้เห็นว่าแนวทางรหัสเงื่อนไขต้องใช้สองคำสั่งสำหรับ ARM-32 และ x86-32 ในขณะที่ RV32I ใช้เพียงคำสั่งเดียว ดังที่บทที่ 2 แสดงในรูปที่ 2.5 ถึง 2.11 แม้ว่า RISC-V จะออกแบบ ISA แบบเรียบง่าย แต่การกระโดดแบบเปรียบเทียบและดำเนินการ (compare-and-execute branches) ลดจำนวนคำสั่งใน Insertion Sort ได้มากเท่ากับโหมดการกำหนดที่อยู่ที่ซับซ้อนและคำสั่ง push และ pop ของ ARM-32 และ x86-32



ประสิทธิภาพ

B.2 การเปรียบเทียบ RV32I, ARM-32 และ x86-32 โดยใช้ Tree Sum

โค้ดภาษา C ในรูปที่ B.4 เป็นตัวอย่างที่เราใช้เปรียบเทียบทั้งสาม ISA แบบเคียงข้างกันในรูปที่ B.5 ถึง B.7 โดยคำนวณผลรวมของค่าในไบนารีทรี โดยใช้การสำรวจทรีแบบตาม

ลำดับ (in-order) ทรีเป็นโครงสร้างข้อมูลพื้นฐาน และแม้ว่าการดำเนินการทรีนี้อาจดูเรียบง่ายเกินไป เราเลือกเพราะมันแสดงทั้งการเรียกซ้ำ (recursion) และการวนซ้ำ (iteration) ในคำสั่งแอสเซมบลีเพียงไม่กี่คำสั่ง โปรแกรมนี้ใช้การเรียกซ้ำเพื่อคำนวณผลรวมของทรีย่อยด้านซ้าย (left subtree) แต่ใช้การวนซ้ำเพื่อคำนวณผลรวมของทรีย่อยด้านขวา (right subtree) ซึ่งลดขนาดหน่วยความจำและจำนวนคำสั่ง คอมไพเลอร์ที่ปรับแต่งแล้วสามารถแปลงโค้ดที่เรียกซ้ำทั้งหมดเป็นรุ่นนี้ได้ เราแสดงการวนซ้ำอย่างชัดเจนเพื่อความชัดเจน

ความแตกต่างที่ใหญ่ที่สุดระหว่างขนาดของโปรแกรมภาษาแอสเซมบลีทั้งสามคือในส่วนการเข้าและออกจากฟังก์ชัน (function entry and exit) RISC-V ใช้สี่คำสั่งเพื่อบันทึกและกู้คืนสามรีจิสเตอร์ในสแตกและปรับตัวชี้สแตก (stack pointer) x86-32 บันทึกและกู้คืนเพียงสองรีจิสเตอร์ในสแตก เพราะสามารถดำเนินการเลขคณิตกับตัวถูกดำเนินการในหน่วยความจำได้โดยไม่ต้องโหลดทั้งหมดลงในรีจิสเตอร์ และยังใช้คำสั่ง push และ pop ซึ่งปรับตัวชี้สแตกโดยนัย (implicitly) แทนการทำอย่างชัดเจนอย่างใน RISC-V ARM-32 สามารถบันทึกสามรีจิสเตอร์รวมถึงลิงก์รีจิสเตอร์ (link register) ที่มีที่อยู่คืนกลับ (return address) ลงในสแตกด้วยคำสั่ง push เดียว และกู้คืนด้วยคำสั่ง pop เดียว RISC-V ดำเนินการลูปหลัก (main loop) ด้วยเจ็ดคำสั่ง แทนแปดคำสั่งใน ISA อื่นๆ เพราะตามที่แสดงในรูปที่ B.3 RISC-V สามารถเปรียบเทียบและกระโดดในคำสั่งเดียว (`bnez`) ในขณะที่ ARM-32 และ x86-32 ต้องใช้สองคำสั่ง คำสั่งที่เหลือในลูปมีการแมปแบบหนึ่งต่อหนึ่งระหว่าง RV32I และ ARM-32 ดังที่แสดงในรูปที่ B.1 และ B.2 ความแตกต่างคือ คำสั่ง `call` และ `ret` ของ x86-32 จะทำการ push และ pop ที่อยู่คืนกลับไปและจากสแตกโดยนัย ในขณะที่ ISA อื่นๆ ทำการบันทึกและกู้คืนอย่างชัดเจนในส่วน prologue และ epilogue (โดยบันทึกและกู้คืน `ra` สำหรับ RV32I หรือ push `lr` และ pop ไปยัง `pc` สำหรับ ARM-32) นอกจากนี้ เนื่องจากข้อกำหนดการเรียกฟังก์ชัน (calling convention) ของ x86-32 ส่งพารามิเตอร์ผ่านสแตก โค้ด x86-32 จึงมีคำสั่ง push และ pop ในลูปที่ ISA อื่นๆ หลีกเลี่ยงได้ การถ่ายโอนข้อมูลเพิ่มเติมนี้ลดประสิทธิภาพ


ประสิทธิภาพ

B.3 บทสรุป

แม้ว่าปรัชญาของ ISA จะแตกต่างกันอย่างมาก แต่โปรแกรมที่ได้นั้นค่อนข้างคล้ายกัน ทำให้การแปลงจากโปรแกรมของสถาปัตยกรรมเก่าไปยัง RISC-V ค่อนข้างตรงไปตรงมา การมี 32 รีจิสเตอร์ใน RISC-V เทียบกับ 16 ใน ARM-32 และ 8 ใน x86-32 ทำให้การถอดความไปยัง RISC-V ง่ายขึ้น ซึ่งจะยากกว่ามากในทิศทางตรงกันข้าม ขั้นตอนการถอดความคือ: ขั้นแรก ปรับส่วน prologue และ epilogue ของฟังก์ชัน จากนั้นเปลี่ยนการกระโดดแบบมีเงื่อนไขจากแบบรหัสเงื่อนไข (condition code oriented) เป็นคำสั่งเปรียบเทียบและกระโดด (compare-and-branch) และสุดท้าย แทนที่ชื่อเรจิสเตอร์และคำสั่งทั้งหมดด้วยเทียบเท่าใน RISC-V อาจมีการปรับแต่งเพิ่มเติม เช่น การจัดการค่าคงที่ขนาดใหญ่และที่อยู่ใน ISA x86-32 ที่มีความยาวแปรผัน หรือการเพิ่มคำสั่ง RISC-V เพื่อรองรับโหมดการกำหนดที่อยู่อันซับซ้อนหากใช้ในการถ่ายโอนข้อมูล แต่การทำตามสามขั้นตอนนี้จะทำให้


การเขียนโปรแกรม

```
struct tree_node {
  struct tree_node *left;
  struct tree_node *right;
  long value;
};

long tree_sum(const struct tree_node *node)
{
  long result = 0;
  while (node) {
    result += tree_sum(node->left);
    result += node->value;
    node = node->right;
  }
  return result;
}
```

รูปที่ B.4: ฟังก์ชัน C ที่รวมค่าต่างๆ ในไบนารีทรี โดยใช้การสำรวจแบบตามลำดับ (in-order traversal)

เข้าใกล้ผลลัพธ์แล้ว

```
    addi sp,sp,-16  # จัดสรร stack frame
    sw   s1,4(sp)   # บันทึก s1
    sw   s0,8(sp)   # บันทึก s0
    sw   ra,12(sp)  # บันทึก ra
    li   s1,0       # sum = 0
    beqz a0,.L1     # Skip loop if node == 0
    mv   s0,a0      # s0 = node
  .L3:
    lw   a0,0(s0)   # a0 = node->left
    jal  tree_sum   # เรียกซ้ำ; result in a0
    lw   a5,8(s0)   # a5 = node->value
    lw   s0,4(s0)   # node = node->right
    add  s1,a0,s1   # sum += a0
    add  s1,s1,a5   # sum += a5
    bnez s0,.L3     # Loop if node != 0
  .L1:
    mv   a0,s1      # Return sum in a0
    lw   s1,4(sp)   # กู้คืน s1
    lw   s0,8(sp)   # กู้คืน s0
    lw   ra,12(sp)  # กู้คืน ra
    addi sp,sp,16   # กลับคืน stack frame
    ret             # คืนค่า
```

รูปที่ B.5: โค้ด RV32I สำหรับการสำรวจทรีแบบตามลำดับ (in-order) ลูปหลักสั้นกว่าใน ISA อื่นๆ เนื่องจากคำสั่งเปรียบเทียบและกระโดด `bnez`

```
  push {r4, r5, r6, lr}  # บันทึก รีจิสเตอร์
  mov  r5, #0            # sum = 0
  subs r4, r0, #0        # r4 = node; node == 0?
  beq  .L1               # Skip loop if so
.L3:
  ldr  r0, [r4]          # r0 = node->left
  bl   tree_sum          # เรียกซ้ำ; result in r0
  ldr  r3, [r4, #8]      # r3 = node->value
  ldr  r4, [r4, #4]      # r4 = node->right
  add  r5, r0, r5        # sum += r0
  add  r5, r5, r3        # sum += r3
  cmp  r4, #0            # node == 0?
  bne  .L3               # Loop if not
.L1:
  mov  r0, r5            # Return sum in r0
  pop  {r4, r5, r6, pc}  # คืนค่ารีจิสเตอร์และส่งคืน
```

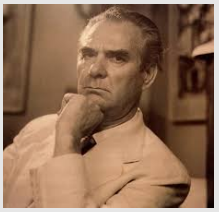
รูปที่ B.6: โค้ด ARM-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) คำสั่ง push และ pop แบบมัลติเวิร์ดช่วยลดขนาดโค้ดสำหรับ ARM-32 เมื่อเทียบกับ ISA อื่นๆ

```
  push esi            # บันทึก esi
  push ebx            # บันทึก ebx
  xor  esi, esi       # sum = 0
  mov  ebx, [esp+12]  # ebx = node
  test ebx, ebx       # node == 0?
  je   .L1            # Skip if so
.L3:
  push [ebx]          # Load node->left; push to stack
  call tree_sum       # เรียกซ้ำ; result in eax
  pop  edx            # Pop old arg and discard
  add  esi, [ebx+8]   # sum += node->value
  mov  ebx, [ebx+4]   # node = node->right
  add  esi, eax       # sum += eax
  test ebx, ebx       # node == 0?
  jne  .L3            # Loop if not
.L1:
  mov  eax, esi       # Return sum in eax
  pop  ebx            # กู้คืน ebx
  pop  esi            # กู้คืน esi
  ret                 # คืนค่า
```

รูปที่ B.7: โค้ด x86-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) ลูปหลักมีคำสั่ง push และ pop ที่ไม่มีในโปรแกรมของ ISA อื่นๆ ซึ่งสร้างการถ่ายโอนข้อมูลเพิ่มเติม

C การแปลภาษาไทย

ศิลป์ พีระศรี (15 กันยายน พ.ศ. 2435–14 พฤษภาคม พ.ศ. 2505) ชื่อ เดิม คอร์ รา โด เฟ โร ซี เป็น ชาว อิตาลี สัญชาติไทย ท่านเป็น ปูชนียบุคคล คน หนึ่ง ของ ไทย โดยได้สร้าง คุณูปการ ในทางศิลปะ จนเป็นที่รู้จัก กว้างขวาง ทั้งยังเป็นผู้ก่อตั้ง มหาวิทยาลัยศิลปากร ได้รับ การยกย่องเป็นปูชนียบุคคล ของมหาวิทยาลัยแห่งนี้ มีผล งานที่โดดเด่นหลายอย่างใน ประเทศไทย

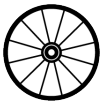


"ศิลปะยืนยาว ชีวิตสั้น" ที่มาจากภาษาละตินว่า "ars longa vita brevis" คำนี้หมายถึง ศิลปะหรือความรู้มีอายุยืนยาวเกินกว่าชีวิตของมนุษย์ ซึ่งสั้นนัก

—ศาสตราจารย์ ศิลป์ พีระศรี

C.1 บทนำ

การแปลคำศัพท์เทคนิคเชิงลึกในบริบทของ ISA ของ RISC-V นั้นต้องใช้ความระมัดระวังในทุกรายละเอียดเพื่อให้ผู้อ่านได้รับข้อมูลที่ถูกต้อง เพราะการเลือกใช้คำแปลที่เหมาะสมเพื่อให้เข้าใจง่ายและรักษาความหมายทางเทคนิค การแปลจึงต้องคำนึงถึงความเฉพาะเจาะจงของ ISA ใช้คำแปลที่สอดคล้องกับนิยามใน RISC-V และ IEEE 754-2019 เช่น จุดลอยตัว แทน จุดทศนิยม การแปลจึงต้องใช้บริบทกำหนดความหมาย เช่น อธิบายว่า "branch" หมายถึงการเปลี่ยนลำดับคำสั่ง จึงใช้คำว่า "การกระโดดแบบมีเงื่อนไข" ในบริบท ISA คำศัพท์ใหม่ เช่น pipeline hazards หรือ strip-mining อาจไม่มีคำแปลที่ตายตัวในภาษาไทย ในทางปฏิบัติ วงการเทคโนโลยีและคอมพิวเตอร์มักใช้คำศัพท์ภาษาอังกฤษ เพื่อให้การสื่อสารชัดเจนและเป็นสากล เนื่องจากเป็นศัพท์เฉพาะที่ผู้อ่านเข้าใจตรงกัน แต่ภาษาไทยอาจต้องใช้คำขยายหรือวลีที่ยาวกว่าเพื่อให้สมบูรณ์ ภาคผนวกนี้ เน้นบางประเด็นที่มีความท้าทายในด้านความเข้าใจ ด้วยความหวังว่าผู้อ่าน จะสามารถเข้าใจถึงความสง่างามและความเรียบง่ายขั้นพื้นฐาน ตามปณิธานที่หนังสือเล่มนี้เขียนไว้แต่แรก



ความเรียบง่าย



ประสิทธิภาพ



การเขียนโปรแกรม

C.2 RISC-V in a Nutshell

คำว่า "in a nutshell" เป็นสำนวนในภาษาอังกฤษที่แปลว่า "โดยย่อ", "อย่างกระชับ", หรือ "ใจความสำคัญ" หมายถึงการสรุปหรืออธิบายบางสิ่ง ที่ซับซ้อนให้เข้าใจง่ายและสั้นที่สุดเท่าที่จะเป็นไปได้ คำว่า "Nutshell" มาจากภาพของการพยายามใส่ข้อมูลจำนวนมากลงในเปลือกถั่ว (nutshell) ซึ่งมีขนาดเล็กมาก "คู่มือสรุปชุดคำสั่ง RISC-V : RISC-V in a Nutshell" สื่อถึงคู่มือที่ให้ภาพรวมของ RISC-V ISA ครอบคลุมใจความสำคัญ เข้าใจง่ายและกระชับ โดยสรุปแนวคิดที่ซับซ้อนให้อยู่ในรูปแบบที่ย่อและชัดเจน เพื่อให้ผู้อ่านเข้าใจพื้นฐานอย่างรวดเร็ว การสรุปสาระสำคัญของ ชุดคำสั่ง RISC-V (Instruction Set

Architecture) รวมถึงส่วนขยาย เช่น Zfh, RV32F, RV32D และการทำงานของจุดลอยตัว (floating-point operations) โดยไม่ต้องอ่านเอกสารเต็ม เช่น The RISC-V Instruction Set Manual เหมาะสำหรับผู้อ่านที่ต้องการเข้าใจภาพรวมของ สถาปัตยกรรมคอมพิวเตอร์หรือชุดคำสั่ง RISC-V โดยไม่ต้องเจาะลึกในรายละเอียดทุกด้าน หนังสือเล่มเล็กนี้จึงเป็นคู่มือหรือเอกสารอ้างอิงสำหรับผู้เริ่มต้น หรือผู้ที่ต้องการทบทวนแนวคิดหลักอย่างรวดเร็ว โดยไม่ลงลึกในรายละเอียดทางเทคนิคที่ซับซ้อนเกินไป

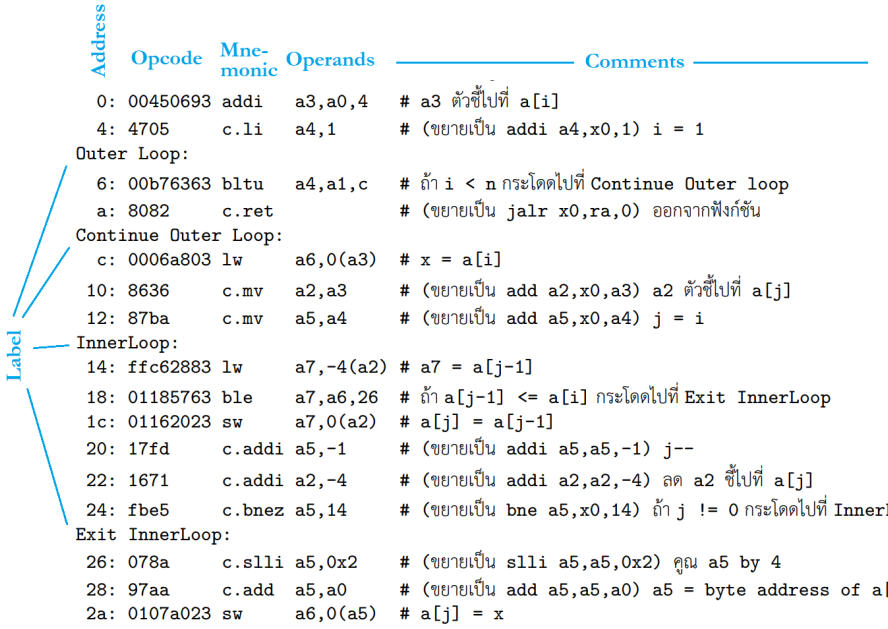
C.3 RISC-V ง่ายเหมือน Pi

RISC-V คือสิ่งใหม่ในวงการ ที่นี่เราจะแสดงวิธีสร้างร้านน้ำมะนาว โดยใช้เพียงสิ่งของประจำวันที่คุณสามารถหาได้ในบ้าน เราจะพาคุณผ่านกระบวนการพื้นฐานของการคอมไพล์, การแอสเซมเบิล, การลิงก์, และการโหลด; อธิบายพื้นฐานว่าการทำงานของ JTAG เป็นอย่างไร (และมันล้มเหลวอย่างไร) และทำทั้งหมดนี้จากความสะดวกสบายของ Raspberry Pi ที่วางทิ้งไว้ให้ฝุ่นเกาะ การคำนวณชุดคำสั่งแบบลดรูป (Reduced Instruction Set Computing) เป็นสิ่งที่เรียบง่าย สมควรได้รับแพลตฟอร์มการพัฒนาที่ลดความซับซ้อนลง RISC-V (RV32I) มีเพียง 47 คำสั่ง, 32 เรจิสเตอร์, และเอกสารประมาณ 236 หน้า เทียบกัน ARM-32 ที่มี 50 คำสั่ง แต่มีรหัสคำสั่ง (opcodes) ที่แตกต่างกันมากกว่า 500 รหัส, มีเพียง 16 เรจิสเตอร์, และเอกสารประมาณ 2,736 หน้า ส่วน x86-32 มี 81 คำสั่ง, มีเพียง 8 เรจิสเตอร์, และเอกสารประมาณ 2,198 หน้า ความรู้พื้นฐานที่ได้จากงานนำเสนอนี้จะเป็นประโยชน์ต่อทั้งต่อผู้เชี่ยวชาญด้านการจัดเก็บข้อมูลและผู้ที่ชื่นชอบคอมพิวเตอร์ไปอีกหลายปี - และสำหรับหลายรูปแบบ – ในอนาคต

C.4 ทำไมต้องใช้ รหัสดำเนินการ (Opcode)?

รหัสดำเนินการ (Opcode) คือส่วนของคำสั่งเครื่อง (machine instruction) ในรูปแบบไบนารีที่ระบุการดำเนินการที่โปรเซสเซอร์ต้องทำ เช่น การบวก (add), การโหลด (load), หรือ การกระโดด (branch) ในโครงร่างของรายการโปรแกรมแอสเซมบลี (AssemblyAnatomy) ซึ่งเป็นการแสดงโครงสร้างของโปรแกรมแอสเซมบลี opcode ถูกนำเสนอควบคู่กับ mnemonic (รหัสช่วยจำ) เพื่อแสดงว่าส่วนของรหัสเครื่องสอดคล้องกับคำสั่งที่มนุษย์อ่านได้อย่างไร สำหรับมนุษย์ เราอ้างถึงคำสั่งโดย “รหัสช่วยจำ (**mnemonic**)” โดยใช้ตัวอักษรที่อ่านได้ง่าย สำหรับคอมพิวเตอร์ คอมพิวเตอร์เข้าถึงคำสั่งโดยใช้ “รหัสดำเนินการ (**op(eration) codes**)” โดยใช้คำที่อ่านง่ายเช่นเดียวกัน เนื่องจากทุกอย่างเกี่ยวกับคอมพิวเตอร์ คือ รหัสดำเนินการ (Opcode) จึงเป็นเพียงตัวเลข ที่มีความเท่าเทียมกันแบบหนึ่งต่อหนึ่งระหว่างรหัสช่วยจำและรหัสดำเนินการ รูปที่ C.1 เป็นตัวอย่างที่แสดงให้เห็นความหมายของ Opcode AssemblyAnatomy แสดงความหมายของ opcode ในโปรแกรมแอสเซมบลีโดย:

- แสดง opcode เป็นรหัสไบนารี/ฐานสิบหกควบคู่กับ mnemonic และ operand



รูปที่ C.1: โครงสร้างโค้ดโปรแกรมแอสเซมบลี แสดงเลเบล (labels), คำอธิบายโค้ด (comments), ที่อยู่หน่วยความจำ (memory address), รหัสดำเนินการ (opcode), รหัสคำสั่ง (mnemonic), และตัวถูกดำเนินการ (operand) หมายเหตุ: รูปแบบการจัดเก็บข้อมูลแบบลำดับไบต์ (endianness) ของระบบที่ใช้จะกำหนดว่าที่อยู่หน่วยความจำที่แสดงนั้นชี้ไปยังไบต์ที่อยู่ด้านขวาสุด (little-endian) หรือด้านซ้ายสุด (big-endian) ของรหัสคำสั่ง (opcode)

เช่น 000130b3 สำหรับ add x1, x2, x3

- เชื่อมโยง opcode กับการทำงานของ CPU เช่น 0110011 ระบุคำสั่ง R-type ใน RV32I
- ระบุที่อยู่หน่วยความจำและคำนึงถึง endianness (เช่น little-endian ใน RISC-V) เพื่อแสดงลำดับไบต์ของ opcode ใช้ร่วมกับเครื่องมือดีบัก เช่น JTAG เพื่อตรวจสอบ opcode ในหน่วยความจำหรือรีจิสเตอร์

โครงร่างของรายการโปรแกรมแอสเซมบลี ทำให้เห็นว่ารหัสดำเนินการ เป็นตัวกำหนดการทำงานของ mnemonic เช่น 0110011 อาจแปลเป็น add, sub, หรือคำสั่งอื่นขึ้นกับฟิลด์ funct3 และ funct7 โดย Opcode ระบุว่า CPU ต้องทำอะไร ในบริบท RISC-V รหัสดำเนินการจึงเป็นหัวใจของคำสั่งเครื่อง ที่กำหนดพฤติกรรมของคำสั่ง

C.5 การขยายบิตเครื่องหมาย

เป็นเทคนิคในระบบคอมพิวเตอร์ที่ใช้ในการขยายขนาดของจำนวนเต็มที่มีเครื่องหมาย

(signed integer) จากความกว้างบิตที่น้อยกว่าไปสู่ความกว้างบิตที่มากขึ้น โดยรักษาค่าและเครื่องหมาย (บวกหรือลบ) ของจำนวนเดิมไว้ ซึ่งมักใช้ในบริบทที่จำนวนอยู่ในรูปแบบ 2's Complement (ส่วนเติมเต็มฐานสอง) ใช้เมื่อต้องแปลงจำนวนที่มีขนาดบิตน้อยกว่าให้เข้ากันได้กับการคำนวณในขนาดบิตที่ใหญ่กว่า เช่น การโหลดข้อมูล 16 บิตไปยังรีจิสเตอร์ 32 บิต ป้องกันการเปลี่ยนแปลงเครื่องหมายหรือค่าของจำนวนเมื่อขยายขนาด ใน RISC-V, การขยายเครื่องหมาย ถูกจัดการโดยฮาร์ดแวร์ในขั้นตอน Decode หรือ Execute ของไปป์ไลน์ ทำให้ไม่ต้องเพิ่มคำสั่งพิเศษ การออกแบบที่เรียบง่ายของ RV32I ทำให้ Sign Extension เป็นส่วนหนึ่งของคำสั่งมาตรฐาน ลดจำนวนคำสั่งพิเศษและเพิ่มความสม่ำเสมอ ทำให้การขยายเครื่องหมายรวดเร็วและไม่เพิ่มความซับซ้อนมากนัก มีบทบาทสำคัญในการจัดการจำนวนเต็มที่มีเครื่องหมายในคำสั่งต่าง ๆ อาทิเช่นการจัดการ immediate ในคำสั่ง (เช่น addi, beq), การโหลดข้อมูลขนาดเล็ก (เช่น lb, lh), และการคำนวณที่อยู่ และส่วนขยายเช่น RV32V (เช่น DAXPY) ช่วยให้การจัดการตัวชี้หน่วยความจำและตัวนับถูกต้อง ซึ่งออกแบบมาให้ทำงานกับรีจิสเตอร์ขนาด 32 บิต (หรือ 64 บิตใน RV64I) คอมไพเลอร์ต้องเลือกคำสั่งที่เหมาะสม (เช่น lb แทน lbu) เพื่อให้ Sign Extension เกิดขึ้นเมื่อจัดการจำนวนที่มีเครื่องหมาย รักษาความถูกต้องของจำนวนที่มีเครื่องหมายและรองรับการคำนวณที่มีจำนวนลบ

C.6 การจัดเรียงลำดับไบต์

การจัดเรียงลำดับไบต์ (Endianness) คือวิธีการกำหนดลำดับไบต์ของข้อมูลในหน่วยความจำ โดย Big-Endian เริ่มจาก MSB ที่ที่อยู่ต่ำสุด และ Little-Endian เริ่มจาก LSB ที่ที่อยู่ต่ำสุด ซึ่งส่งผลต่อการแสดงที่อยู่ในหน่วยความจำและการเข้าถึงข้อมูล การทำความเข้าใจและจัดการกับ Endianness ให้เหมาะสมจึงเป็นสิ่งสำคัญในการพัฒนาซอฟต์แวร์และการสื่อสารข้อมูลในระบบคอมพิวเตอร์ ในระบบ Big-Endian การเข้าถึงข้อมูลจะเริ่มจากไบต์ที่มีน้ำหนักมากที่สุด (MSB) ก่อน ซึ่งสอดคล้องกับลำดับการเขียนตัวเลขตามปกติ (จากซ้ายไปขวา) หรืออ่านแบบคนไทยอ่านหนังสือในระบบ Little-Endian การเข้าถึงข้อมูลจะเริ่มจากไบต์ที่มีน้ำหนักน้อยที่สุด (LSB) ก่อน ซึ่งอาจดูไม่เป็นธรรมชาติ แต่มีข้อดีในด้านการคำนวณทางคณิตศาสตร์บางประเภท เช่น การบวกเลขที่มีความยาวต่างกัน ภาษาคอมพิวเตอร์จะมีโครงสร้างคล้ายกัน ขึ้นอยู่กับอุปกรณ์ฮาร์ดแวร์ที่ใช้ จำกัดภาษาระดับต่ำไว้เพียงคำสั่งพื้นฐานเพื่อให้สร้างได้ง่ายขึ้น ลดค่าใช้จ่าย และประหยัดทรัพยากรมากขึ้น แต่ยังคงทำงานอย่างมีประสิทธิภาพสูงสุด ภาษาที่คอมพิวเตอร์เข้าใจคือลำดับของเลข 0 และ 1 เราสามารถเรียงลำดับของตัวเลขจาก ซ้ายไปขวาหรือจากขวาไปซ้ายได้ ซึ่งการอ่านในแต่ละทิศทางสามารถแปลความหมายได้ต่างกัน ดังนั้นเราจึงมีการกำหนดการเข้าถึงตำแหน่งของไบต์ (byte addressing) เพื่อกำหนดลำดับของไบต์ (byte order) ในทางสถาปัตยกรรมคอมพิวเตอร์ คำว่า “เอนเดียน” (endian) หมายถึง วิธีที่คอมพิวเตอร์จัดเก็บไบต์ (“byte order” เป็นแนวคิดในระบบคอมพิวเตอร์ที่กำหนดลำดับการจัดเก็บไบต์ของข้อมูลที่มีขนาดหลายไบต์ (เช่น ค่าตัวเลข 16 บิต หรือ 32 บิต) ในหน่วยความจำ โดย

ในระบบดิจิทัล ข้อมูลที่มีขนาดใหญ่กว่าหนึ่งไบต์จะถูกแบ่งออกเป็นไบต์ย่อย ๆ และลำดับที่ไบต์เหล่านี้ถูกจัดเก็บจะขึ้นอยู่กับรูปแบบของ Endianness

C.7 รายการตัวย่อ

ABI Application Binary Interface: ส่วนต่อประสานระบบปฏิบัติการกับส่วนผู้ใช้ของชุดคำสั่งที่ใช้โดยโปรแกรมเมอร์แอปพลิเคชัน
AMO Atomic Memory Operations: การดำเนินการหน่วยความจำอะตอมิก
API Application Program Interface: เป็นวิธีสำหรับโปรแกรมคอมพิวเตอร์หรือส่วนประกอบตั้งแต่สองโปรแกรมขึ้นไปในการสื่อสารระหว่างกัน
ASICs Application-Specific Integrated Circuit: เป็นชิปวงจรรวม (IC) ที่ปรับแต่งสำหรับการใช้งานเฉพาะ
BCD Binary Coded Decimal: เป็นคลาสของการเข้ารหัสไบนารี่ของเลขทศนิยม โดยแต่ละหลักจะแสดงด้วยจำนวนบิตคงที่ โดยปกติจะเป็นสี่หรือแปด
CISC Complex Instruction Set Computer: คอมพิวเตอร์ชุดคำสั่งซับซ้อน
CPI Cycles Per Instruction: จำนวนรอบสัญญาณนาฬิกาเฉลี่ยต่อคำสั่งสำหรับโปรแกรม
CSR Control and Status Registers: รีจิสเตอร์ควบคุมและสถานะ
DAXPY Double-precision AX Plus Y: อัลกอริทึมสำหรับการคำนวณเวกเตอร์
ELF Executable and Linkable Format: เป็นรูปแบบไฟล์มาตรฐานทั่วไปสำหรับไฟล์ปฏิบัติการ รหัสอ็อบเจ็กต์ ไลบรารีที่ใช้ร่วมกัน
FPD field programmable devices: วงจรรวมที่มีตรรกะเชิงผสมและอุปกรณ์หน่วยความจำที่กำหนดได้โดยผู้ใช้ปลายทาง
FPGAs Field-Programmable Gate Arrays: วงจรรวมที่กำหนดค่าได้ เป็นอุปกรณ์ไอซีที่ผู้ใช้สามารถทำการโปรแกรมวงจรภายในได้เอง
FPUs Floating Point Units: เป็นส่วนหนึ่งของระบบคอมพิวเตอร์ที่ออกแบบมาเป็นพิเศษเพื่อดำเนินการกับจำนวนจุดทศนิยม
LMUL Length Multiplier: ตัวคูณความยาวเวกเตอร์ ใน RV32V
GDB GNU Debugger: เครื่องมือดีบัก
GP Global Pointer: ตัวชี้ส่วนกลาง
PC Stack Pointer: รีจิสเตอร์เก็บที่อยู่คำสั่ง
Hart Hardware thread: หน่วยวัดจำนวนสัญญาณนาฬิกาต่อวินาที
MFLOPS Mega floating-point operations per second: เป็นหน่วยวัดประสิทธิภาพของคอมพิวเตอร์ในด้านการคำนวณ ซึ่งมีประโยชน์ในด้านการคำนวณทางวิทยาศาสตร์ที่ต้องใช้การคำนวณจุดลอยตัว
IMM Immediate: ข้อมูลค่าคงที่
IPI Inter-Processor Interrupt: ประสานงานระหว่าง hart
ISA Instruction Set Architecture: สถาปัตยกรรมชุดคำสั่ง
MARCHID Machine Architecture ID: รหัสสถาปัตยกรรมเครื่อง
MHARTID Machine Hardware Thread ID: ตัวระบุคอร์ที่กำลังดำเนินงาน

MIE Machine mode Interrupt Enable: เปิดใช้งานการขัดจังหวะ

MIP Machine mode Interrupt Pending: การขอขัดจังหวะรอดำเนินการ

NOP Program Counter: รีจิสเตอร์เก็บที่อยู่คำสั่ง

PC No Operation: คำสั่งที่ไม่ทำอะไร

RISC Reduced Instruction Set Computer: คอมพิวเตอร์ชุดคำสั่งลดลง

RISC-I Reduced Instruction Set Computer 1st major design: Invented by David Patterson in 1980 at University of California Berkeley. Called GOLD, it is quite likely based on ideas of John Cocke and the IBM 801 a few years earlier.

RISC-II Reduced Instruction Set Computer 2nd major design: Berkeley project called BLUE.

RISC-III Reduced Instruction Set Computer 3rd major design: Berkeley project called SOAR (Smalltalk on a RISC) to make Smalltalk run better.

RISC-IV Reduced Instruction Set Computer 4th major design: Berkeley project called SPUR (Symbolic Processing Using RISCs) to create a multi-processor workstation running conventional programs and Lisp.

RISC-V Reduced Instruction Set Computer 5th major design: สถาปัตยกรรมชุดคำสั่งแบบเปิดที่ใช้หลักการคอมพิวเตอร์ชุดคำสั่งแบบลดคำสั่ง พัฒนาใน 2010 โดย UC Berkeley เพื่อให้สามารถปรับแต่งและใช้งานได้ฟรีโดยไม่ต้องเสียค่าลิขสิทธิ์

RPM Revolution Per Minute: รอบการทำงานของคำสั่งต่อนาที

SBI Supervisor Binary Interface: ส่วนต่อประสานมาตรฐานในสถาปัตยกรรม RISC-V

SP Stack Pointer: ตัวชี้สแต็ก

SIMD Single Instruction Multiple Data: คำสั่งเดียวใช้กับสตรีมข้อมูลจำนวนมาก

SISD Single Instruction stream, Single Data stream: คำสั่งเดียวสตรีมข้อมูลเดียว

SPLD simple programmable logic device: อุปกรณ์ลอจิกแบบตั้งโปรแกรมอย่างง่าย

SEW Standard Element Width: ความกว้างสมาชิกเวกเตอร์

XLEN Instruct (X) Length: ขนาดความกว้างของรีจิสเตอร์ X บิต

VLSI very-large-scale integrated: อุปกรณ์ที่มีทรานซิสเตอร์หลายแสนตัวถึงหลายล้านตัวคำคำสั่งที่ยาวมาก (VLIW) รูปแบบของ สถาปัตยกรรมชุดคำสั่ง ที่เรียกใช้การดำเนินการจำนวนมาก

LMUL Length Multiplier: ตัวคูณความยาวรีจิสเตอร์เวกเตอร์ใน RV32V

SEW Standard Element Width: ความกว้างสมาชิกเวกเตอร์ใน RV32V

vl Vector Length: ความยาวเวกเตอร์ใน RV32V

C.8 ความสอดคล้องของผู้แปล

Technical Terms:

-
- XLEN: คำย่อที่ใช้ในเอกสาร RISC-V เพื่อแสดงถึงความกว้างของรีจิสเตอร์จำนวนเต็ม

(integer register width) ซึ่งขึ้นอยู่กับรุ่นของสถาปัตยกรรม

- hart: ย่อมาจาก "hardware thread" หมายถึงหน่วยประมวลผลระดับฮาร์ดแวร์ใน RISC-V
- Software threads: เธรดที่ทำงานในระดับซอฟต์แวร์ ซึ่งถูกจัดการโดยระบบปฏิบัติการและแบ่งเวลาการใช้งานบน hart
- cache memory หน่วยความจำขนาดเล็ก เป็นหน่วยความจำที่รวดเร็วซึ่งทำหน้าที่เป็นบัฟเฟอร์
- cache miss คำขอข้อมูลจากแคชที่ไม่สามารถกรอกได้เนื่องจากไม่มีข้อมูลอยู่ในแคช
- D flip-flop ฟลิปฟลอปที่มีการป้อนข้อมูลหนึ่งรายการที่เก็บค่าของสัญญาณอินพุตนั้น ในหน่วยความจำภายในเมื่อข บนาฬิกาเกิดขึ้น
- gate อุปกรณ์ที่ใช้ฟังก์ชันลอจิกพื้นฐาน เช่น AND หรือ OR
- flip-flop องค์ประกอบหน่วยความจำที่เอาต์พุตเท่ากับ ค่าของสถานะที่เก็บไว้ภายในองค์ประกอบ และสถานะภายในจะถูกเปลี่ยนเฉพาะบนขอบนาฬิกาเท่านั้น
- frame pointer ค่าที่แสดงถึงตำแหน่งของรีจิสเตอร์ที่บันทึกไว้ และตัวแปรโลคัลสำหรับโพรซีเดอร์ที่กำหนด
- fused multiply add คำสั่งทศนิยมที่ดำเนินการทั้งการคูณและการบวก แต่จะปัดเพียงครั้งเดียวหลังจากการบวก
- Figure รูปที่
- synchronization การประสานของกระบวนการตั้งแต่สองกระบวนการขึ้นไป ซึ่งอาจทำงานอยู่บนโปรเซสเซอร์ที่แตกต่างกัน
- Synchronous exceptions: ข้อยกเว้นที่เกิดจากคำสั่งในโปรแกรม เช่น การเข้าถึงหน่วยความจำที่ไม่ถูกต้อง
- system call คำสั่งพิเศษที่ ถ่ายโอนการควบคุมจากโหมดผู้ใช้ไปยังตำแหน่งเฉพาะในพื้นที่โค้ดของผู้ควบคุมโดยเรียกใช้กลไกข้อยกเว้นในกระบวนการ
- Endianness: หมายถึงลำดับการจัดเก็บไบต์ของข้อมูลในหน่วยความจำ
- lLittle-endian: ไบต์ที่มีความสำคัญน้อยที่สุด (least significant byte) ถูกเก็บที่ที่อยู่ต่ำสุด ตัวอย่าง: ค่า 0x12345678 จะถูกเก็บเป็น 78 56 34 12
- Big-endian: ไบต์ที่มีความสำคัญมากที่สุด (most significant byte) ถูกเก็บที่ที่อยู่ต่ำสุด ตัวอย่าง: ค่า 0x12345678 จะถูกเก็บเป็น 12 34 56 78

- XOR-swap: เทคนิคที่ใช้คำสั่ง xor เพื่อสลับค่าของสองรีจิสเตอร์โดยไม่ต้องใช้รีจิสเตอร์เพิ่มเติม
- PC (Program Counter) เป็นหนึ่งในรีจิสเตอร์ใน ARM-32 ทำให้คำสั่งที่เปลี่ยนรีจิสเตอร์
- M-mode: โหมดเครื่อง (Machine Mode) เป็นโหมดที่มีสิทธิพิเศษสูงสุดใน RISC-V ใช้สำหรับการจัดการระบบระดับต่ำ เช่น การบูตและการตั้งค่า
- mcause: รีจิสเตอร์ใน RISC-V ที่ใช้เก็บข้อมูลเกี่ยวกับสาเหตุของข้อยกเว้นหรือการขัดจังหวะ
- Synchronous exceptions: ข้อยกเว้นที่เกิดจากคำสั่งในโปรแกรม เช่น การเข้าถึงหน่วยความจำที่ไม่ถูกต้อง
- Interrupts: เหตุการณ์ที่เกิดจากภายนอก เช่น อุปกรณ์ I/O ส่งสัญญาณ
- Precise exceptions: คุณสมบัติที่รับประกันว่าสถานะของโปรแกรมจะคงที่เมื่อเกิดข้อยกเว้น ทำให้สามารถจัดการและกู้คืนได้อย่างแม่นยำ
- Opcode: รหัสคำสั่งที่ใช้ในระดับเครื่อง (machine code) เพื่อระบุการทำงานของคำสั่งนั้นๆ
- Concluding Remarks ข้อสังเกตสรุป
- To Learn More เรียนรู้เพิ่มเติม
- Appendix ภาคผนวก
- Chapter บทที่
- Section เรื่องที่
- Page หน้าที่
- Figure รูปที่
- Table ตารางที่
- Introduction บทนำ
- Concluding Remarks ข้อสังเกตสรุป
- To Learn More เรียนรู้เพิ่มเติม
- Appendix ภาคผนวก
- Chapter บทที่

- Section เรื่องที่
- Page หน้าที่
- Figure รูปที่
- Table ตารางที่
- Introduction บทนำ
- Concluding Remarks ข้อสังเกตสรุป
- To Learn More เรียนรู้เพิ่มเติม
- Appendix ภาคผนวก
- Chapter บทที่
- Section เรื่องที่
- Page หน้าที่
- Figure รูปที่
- Table ตารางที่
- Introduction บทนำ
- Concluding Remarks ข้อสังเกตสรุป
- To Learn More เรียนรู้เพิ่มเติม
- Appendix ภาคผนวก
- Chapter บทที่
- Section เรื่องที่
- Page หน้าที่
- Figure รูปที่
- Table ตารางที่
- Introduction บทนำ
- Concluding Remarks ข้อสังเกตสรุป
- To Learn More เรียนรู้เพิ่มเติม
- Appendix ภาคผนวก

Learning Terms:

- Opcode: รหัสคำสั่งที่ใช้ในระดับเครื่อง (machine code) เพื่อระบุการทำงานของคำสั่งนั้นๆ
-
- absolute address ที่อยู่จริงของตัวแปรหรือรูทีนในหน่วยความจำ
- access bit บิตการเข้าถึง เรียกอีกอย่างว่าบิตใช้ หรือบิตอ้างอิงฟิลด์ ที่ตั้งค่าทุกครั้งที่มีการเข้าถึงเพจ และใช้เพื่อนำ LRU ไปใช้ งานหรือโครงร่างการแทนที่อื่นๆ
- active matrix display การแปลที่อยู่ เรียกอีกอย่างว่าการทำแผนที่ที่อยู่ กระบวนการที่แมปที่อยู่เสมือนกับที่อยู่ที่ใช้ในการเข้าถึงหน่วยความจำ
- address mode โหมดการกำหนดตำแหน่งที่อยู่
- Amdahl's Law กฎของ Amdahl กฎที่ระบุว่าการเพิ่มประสิทธิภาพที่เป็นไปได้ ด้วยการปรับปรุงที่กำหนดนั้น ถูกจำกัดด้วยจำนวนที่ใช้คุณลักษณะที่ปรับปรุงแล้ว เป็นเวอร์ชันเชิงปริมาณของกฎว่าด้วยผลตอบแทนที่ลดลง
- architectural registers รีจิสเตอร์ที่มองเห็นได้ของโปรเซสเซอร์
- assembler โปรแกรมที่แปลเวอร์ชันสัญลักษณ์ของคำสั่งเป็นเวอร์ชันไบนารี
- assembler directive การดำเนินการที่บอกแอสเซมเบลอร์ถึงวิธีการแปลโปรแกรม แต่ไม่สร้างคำสั่ง เครื่องเริ่มต้นด้วยจุดเสมอ
- benchmark โปรแกรมที่เลือกใช้ในการเปรียบ เทียบประสิทธิภาพของคอมพิวเตอร์
- binary digit เรียกอีกอย่างว่าบิตไบนารี หนึ่งในสองตัวเลขในฐาน 2, 0 หรือ1 ซึ่งเป็นส่วนประกอบของข้อมูล
- branch address table ตารางตำแหน่งที่อยู่ของลำดับคำสั่งทางเลือก
- branch prediction วิธีการแก้ไขอันตรายของการบรานช์ ที่ถือว่าผลลัพธ์ที่กำหนดสำหรับบรานช์แบบมีเงื่อนไขและดำเนินการจากสมมติฐานนั้น แทนที่จะรอเพื่อยืนยันผลลัพธ์จริง
- bus ชุดของสายข้อมูล ที่ถือว่าเป็นสัญญาณลอจิคัลเดียว
- callee โพรซีเดอร์ที่รันชุดคำสั่งที่เก็บไว้ตามพารามิเตอร์ที่ผู้เรียกให้มา
- callee-saved register รีจิสเตอร์ที่บันทึกโดยรูทีนในการเรียกโพรซีเดอร์
- caller โปรแกรมที่กระตุ้น โพรซีเดอร์และจัดเตรียมค่าพารามิเตอร์ที่จำเป็น
- caller-saved register รีจิสเตอร์ที่บันทึกผู้โทร รีจิสเตอร์ที่บันทึกโดยรูทีนที่ถูกเรียก
- clock cycle เวลาสำหรับช่วงสัญญาณนาฬิกาหนึ่งช่วงโดยปกติของนาฬิกาของโปรเซสเซอร์

- clock cycles per instruction (CPI) รอบสัญญาณนาฬิกาต่อคำสั่ง (CPI) จำนวนรอบสัญญาณนาฬิกาเฉลี่ยต่อคำสั่งสำหรับโปรแกรมหรือส่วนย่อยของโปรแกรม
- clock period ความยาวของรอบนาฬิกาแต่ละรอบ
- die ส่วนสี่เหลี่ยมแต่ละส่วนซึ่งถูกตัดจากแผ่นเวเฟอร์ หรือที่รู้จักกันอย่างไม่เป็นทางการว่าชิป
- double precision ความแม่นยำสองเท่าของค่าจุดทศนิยมที่แสดงในดับเบิลเวิร์ด 64 บิต
- doubleword หน่วยของการเข้าถึงข้อมูลในคอมพิวเตอร์ปกติแล้วจะเป็นกลุ่ม 64 บิต
- dynamic branch prediction การทำนายบรานช์ที่รันไทม์โดยใช้ข้อมูลรันไทม์
- dynamically linked libraries (DLLs) ไลบรารีที่เชื่อมโยงแบบไดนามิก (DLLs) รูทีนของไลบรารีที่เชื่อมโยงกับโปรแกรมระหว่างการดำเนินการ
-
- data-level parallelism ความขนานระดับข้อมูลความขนานที่ทำได้โดยการ ดำเนินการเดียวกันกับข้อมูลอิสระ
- datapath ส่วนประกอบของโปรเซสเซอร์ที่ทำการคำนวณทางคณิตศาสตร์
- datapath element องค์ประกอบ datapath หน่วยที่ใช้ในการดำเนินการหรือเก็บข้อมูลภายในตัวประมวลผล
- decoder ตัวถอดรหัสบล็อกลอจิกที่มีอินพุตแบบ n-bit และเอาต์พุต2n โดยจะมีการยืนยันเอาต์พุตเพียงรายการเดียวสำหรับชุดอินพุตและชุดข้อบกพร่อง
- delayed branch ประเภทของบรานช์ที่ดำเนินการคำสั่งทันที โดยไม่ขึ้นกับว่าเงื่อนไขของสาขาเป็นจริงหรือเท็จ
- embedded computer คอมพิวเตอร์ระบบฝังตัวภายในอุปกรณ์อื่น ที่ใช้สำหรับเรียกใช้แอพพลิเคชัน หรือชุดซอฟต์แวร์ที่กำหนดไว้ล่วงหน้า
- exception การขัดจังหวะเหตุการณ์ที่ไม่ได้กำหนดไว้ ซึ่งขัดขวางการทำงานของโปรแกรม ใช้ในการตรวจจับโอเวอร์โฟล์
- exception enable เรียกอีกอย่างว่าเปิดใช้งานการขัดจังหวะสัญญาณ หรือการกระทำที่ควบคุม ว่ากระบวนการตอบสนองต่อข้อยกเว้น หรือไม่จำเป็นสำหรับการป้องกันการเกิดข้อยกเว้นในช่วงเวลาก่อนที่ โปรเซสเซอร์จะบันทึกสถานะที่จำเป็นในการรีสตาร์ทอย่างปลอดภัย
-
- global miss rate เศษส่วนของการอ้างอิงที่ขาดหายไปในทุกระดับ ของแคชหลายระดับ

- global pointer รีจิสเตอร์ที่สงวนไว้เพื่อชี้ไปยังพื้นที่สแตติก
- guard ป้องกันบิตพิเศษสองบิตแรก เก็บไว้ทางด้านขวา ระหว่างการคำนวณตัวเลขทศนิยม เพื่อปรับปรุงความแม่นยำ ในการปัดเศษ
- handler ตัว จัดการ ชื่อ ของ รู ทีน ซอฟต์แวร์ ที่ เรียก ใช้ เพื่อ"จัดการ" ข้อ ยกเว้น หรือ ขัดจังหวะ
- hardware multithreading เพิ่มการใช้โปรเซสเซอร์โดยสลับไปยังเธรดอื่น เมื่อเธรดหนึ่งหยุดทำงาน
- ecall (Environment Call): ใช้เรียกบริการจากสภาพแวดล้อม เช่น การเรียกระบบในระบบปฏิบัติการ (system calls) เพื่อขอทรัพยากรหรือบริการ
- ebreak (Environment Break): ใช้ในการดีบัก โดยหยุดการทำงานของโปรแกรมและถ่ายโอนการควบคุมไปยังดีบักเกอร์
- fence (Fence): ควบคุมลำดับการเข้าถึงหน่วยความจำและ I/O เพื่อให้แน่ใจว่าเธรดอื่นหรืออุปกรณ์ภายนอกเห็นการเปลี่ยนแปลงในลำดับที่ถูกต้อง
- hit rate เศษของการเข้าถึงหน่วยความจำ ที่พบในระดับของลำดับชั้นหน่วยความจำ
- hit time เวลาที่ต้องใช้ในการเข้าถึงระดับของลำดับชั้นหน่วยความจำ รวมถึงเวลาที่จำเป็นในการพิจารณาว่าการเข้าถึงนั้นเป็น hit หรือ miss
- hold time เวลาต่ำสุดระหว่างที่อินพุต ต้องถูกต้องหลังจากขอบนาฬิกาการ
- imprecise interrupt ขัดจังหวะไม่ แน่ชัดเรียกอีกอย่างว่าข้อยกเว้นที่ไม่แน่ชัด การขัดจังหวะ หรือ ข้อ ยกเว้น ใน คอมพิวเตอร์ แบบ ไปป์ ไลน์ ที่ ไม่ เกี่ยวข้อง กับ คำ สั่ง ที่แน่นอนซึ่งเป็นสาเหตุของการขัดจังหวะหรือข้อยกเว้น
- implementation การนำฮาร์ดแวร์ไปใช้ที่เป็นไปตามนามธรรมของสถาปัตยกรรม
- input device อุปกรณ์อินพุตกลไกที่คอมพิวเตอรับข้อมูล์
- instruction คำสั่งคำสั่งที่ฮาร์ดแวร์คอมพิวเตอร์เข้าใจและปฏิบัติตาม
- instruction count จำนวนคำสั่งจำนวนคำสั่งที่ดำเนินการโดยโปรแกรม
- instruction format รูปแบบคำสั่งรูปแบบการแสดงคำสั่งที่ประกอบด้วยช่อง เลขฐานสอง
- instruction-level parallelism ความขนานระดับคำสั่งความ ขนานระหว่างคำสั่งคำสั่งผสม การวัดความถี่ไดนามิกของคำสั่ง ในโปรแกรมเดียวหรือหลายโปรแกรม
- instruction set architecture สถาปัตยกรรม ชุด คำ สั่ง เรียก อีก อย่าง ว่า สถาปัตยกรรมอินเทอร์เฟซแบบนามธรรม ระหว่างฮาร์ดแวร์และ ซอฟต์แวระดับต่ำสุด ที่รวมข้อมูลทั้งหมดที่จำเป็นในการเขียนโปรแกรมภาษาเครื่อง ที่จะทำงานอย่างถูกต้อง รวมถึงคำสั่งรีจีสเตอร์ การเข้าถึงหน่วยความจำ I/O และอื่นๆ

- Interrupts: ขัดจังหวะข้อยกเว้นที่เกิดจากภายนอกโปรเซสเซอร์ เช่น อุปกรณ์ I/O ส่งสัญญาณ
- interrupt handler ตัวจัดการการขัดจังหวะส่วนของรหัส ที่รันอันเป็นผลมาจากข้อยกเว้น หรือการขัดจังหวะ
- loader โปรแกรมระบบที่วางโปรแกรม อ็อบเจ็กต์ในหน่วยความจำหลักเพื่อให้พร้อมที่จะดำเนินการ
- latch องค์ประกอบหน่วยความจำ ที่เอาต์พุตเท่ากับค่าของสถานะที่เก็บไว้ภายในองค์ประกอบ และสถานะจะเปลี่ยนแปลงทุกครั้งที่อินพุตที่เหมาะสมเปลี่ยนและนาฬิกาถูกยืนยัน
- latency เวลาแฝง (ไปป์ไลน์) จำนวนขั้นตอนในไปป์ไลน์หรือจำนวนขั้นตอนระหว่างสองคำสั่งระหว่างการดำเนินการ
- least significant bit บิตที่มีนัยสำคัญน้อยที่สุดบิตขวาสุดใน doubleword RISC-V
- linker เรียกอีกอย่างว่าตัวแก้ไขลิงค์โปรแกรมระบบ ที่รวมโปรแกรมภาษาเครื่อง ที่ประกอบกันอย่างอิสระและแก้ไขเลเบล ที่ไม่ได้กำหนดทั้งหมดให้เป็นไฟล์เรียกทำงาน
- loader โปรแกรมระบบที่วางโปรแกรมอ็อบเจ็กต์ในหน่วยความจำหลัก เพื่อให้พร้อมที่จะดำเนินการ
- local label เลเบลที่อ้างถึงอ็อบเจ็กต์ที่ สามารถใช้ได้เฉพาะภายในไฟล์ที่กำหนดไว้เท่านั้น
- lock อุปกรณ์ซิงโครไนซ์ที่อนุญาตให้เข้าถึงข้อมูลไปยังโปรเซสเซอร์ได้ ครั้งละหนึ่งตัวเท่านั้น
- mcause: รีจิสเตอร์ใน RISC-V ที่ใช้เก็บข้อมูลเกี่ยวกับสาเหตุของข้อยกเว้นหรือการขัดจังหวะ
- MIMD หรือสตรีมคำสั่งหลายรายการ สตรีมข้อมูลหลายรายการ มัลติโปรเซสเซอร์
- most significant bit บิตที่สำคัญที่สุดบิตซ้ายสุดใน doubleword RISC-V
- nop คำสั่งที่ไม่มีการดำเนินการเพื่อเปลี่ยนสถานะ
- out-of-order execution การดำเนินการที่ไม่อยู่ในลำดับสถานการณ์ ในการดำเนินการไปพ์ไลน์ เมื่อคำสั่งที่ถูกบล็อกไม่ให้ดำเนินการไม่ทำให้คำสั่งต่อไปนี้รอ
- opcode ฟิลด์ที่แสดงถึงการดำเนินการและรูปแบบของคำสั่ง
- parallel processing program โปรแกรมประมวล ผลแบบขนาน โปรแกรมเดียวที่ทำงานบนโปรเซสเซอร์หลายตัวพร้อมกัน

- PC-relative addressing การกำหนดแอดเดรสแบบสัมพันธ์กับพีซี ระบบการกำหนดแอดเดรสที่ตำแหน่งที่อยู่เป็นผลรวมของตัวนับโปรแกรม (PC) และค่าคงที่ในคำสั่ง
- physical address ที่อยู่ทางกายภาพ
- pipelining เทคนิคการนำไปใช้โดยที่คำสั่งหลายคำสั่งซ้อนทับกัน ในการดำเนินการเหมือนกับสายการประกอบ
- pop ลบองค์ประกอบออกจากสแต็ก
- procedure รูทีนย่อยที่เก็บไว้ซึ่งทำงานเฉพาะตามพารามิเตอร์ ที่มีให้เฟรมการเรียกโพรซีเดอร์
- Precise exceptions: คุณสมบัติที่รับประกันว่าสถานะของโปรแกรมจะคงที่เมื่อเกิดข้อยกเว้น ทำให้สามารถจัดการและกู้คืนได้อย่างแม่นยำ
- program counter (PC) รีจิสเตอร์ที่มีตำแหน่งที่อยู่ของคำสั่งในโปรแกรมที่กำลังดำเนินการ
- pseudo instruction รูปแบบทั่วไปของคำสั่ง ภาษาแอสเซมบลีมักจะถูกปฏิบัติราวกับว่ามันเป็นคำสั่งในตัวของมันเอง
- Push เพิ่มองค์ประกอบเพื่อสแต็ค
- parallel processing program โปรแกรมประมวลผลแบบขนาน โปรแกรมเดียวที่ทำงานบนโปรเซสเซอร์หลายตัวพร้อมกัน
- quotient ผลหารคือจำนวนที่เมื่อคูณด้วยตัวหารและบวกกับส่วนที่เหลือ ทำให้ได้ตัวตั้งหาร (dividend)
- Race Condition: สถานการณ์ที่หลายเธรดเข้าถึงหน่วยความจำร่วมกันและทำให้ผลลัพธ์ไม่แน่นอน
- round ปัดเศษวิธีการทำผลทศนิยมกลางให้พอดีกับรูปแบบจุดลอยตัว เป้าหมายโดยทั่วไปคือการหาตัวเลขที่ใกล้ที่สุด
- scientific notation สัญกรณ์วิทยาศาสตร์ สัญกรณ์ที่แสดงตัวเลขที่มีหลักเดียวทางด้านซ้ายของจุดทศนิยม
- sign-extend การขยายเครื่องหมายของค่าแบบทันที โดยใช้บิตเครื่องหมายที่ตำแหน่งคงที่ช่วยให้การประมวลผลเร็วขึ้น โดยไม่ต้องรอถอดรหัสคำสั่ง
- SIMD หรือสตรีมคำสั่งเดียวสตรีมข้อมูล หลายรายการ คำสั่งเดียวกันนี้ใช้กับสตรีมข้อมูลจำนวนมาก
- single precision ค่าทศนิยมที่แสดงในเวิร์ดแบบ 32 บิต
- stack โครงสร้างข้อมูลสำหรับการลงทะเบียนการรั่วไหลที่จัดเป็นคิวสุดท้ายก่อนออก

- stack pointer ตัวชี้สแต็กค่าที่แสดง ถึงที่อยู่ที่จัดสรรล่าสุดในสแต็กที่แสดงตำแหน่งที่รีจิสเตอร์ควรรั่วไหล หรือตำแหน่งที่สามารถหาค่ารีจิสเตอร์เก่าได้ใน RISC-V คือ register x2 หรือที่เรียกว่า sp
- stack segment ส่วนของหน่วยความจำที่โปรแกรมใช้เพื่อเก็บโพรซีเดอร์ call frames องค์ประกอบสถานะ
- supercomputer คลาสของคอมพิวเตอร์ที่มีการปรับขนาดที่แข็งแกร่ง การเพิ่มความเร็วทำได้บนมัลติโปรเซสเซอร์
- superscalar เทคนิคการ pipe lline ขั้นสูงที่ช่วยให้โปรเซสเซอร์สามารถดำเนินการได้มากกว่าหนึ่งคำสั่งต่อรอบสัญญาณนาฬิกา
- Synchronization:: การประสานงานระหว่างเธรดเพื่อให้แน่ใจว่าการเข้าถึงหน่วยความจำเกิดขึ้นอย่างถูกต้อง
- Illegal instructions: รูปแบบบิตที่เป็นศูนย์ทั้งหมดหรือหนึ่งทั้งหมดถูกกำหนดให้เป็นคำสั่งที่ไม่ถูกต้องใน RV32I เพื่อช่วยตรวจจับข้อผิดพลาด
- scientific notation สัญกรณ์วิทยาศาสตร์สัญกรณ์ที่แสดงตัวเลขที่มี หลักเดียวทางด้านซ้ายของจุดทศนิยม
- truth table ตารางความจริงจากตรรกะ การแสดงการดำเนินการทางตรรกะ โดยแสดงรายการค่าทั้งหมดของอินพุตจากนั้น ในแต่ละกรณีจะแสดงว่าผลลัพธ์ ที่ได้ควรเป็นอย่างไร
- throughput ปริมาณงาน เรียกอีกอย่างว่าแบนด์วิดธ์ การวัดประสิทธิภาพอีกอย่างคือ จำนวนงานที่เสร็จสิ้นต่อหน่วยเวลา
- underflow (floating-point) อันเดอร์โฟลว์(จุดทศนิยม) สถานการณ์ที่เลขชี้กำลังลบมีขนาดใหญ่เกินกว่าจะใส่ลงในช่องเลขชี้กำลัง
- valid bit บิตที่ถูกต้องฟิลด์ในตารางของลำดับชั้นหน่วยความจำ ที่ระบุว่าบล็อกที่เกี่ยวข้องในลำดับชั้น ประกอบด้วยข้อมูลที่ถูกต้อง
- virtual addressx ที่อยู่เสมือนที่สอดคล้องกับตำแหน่งในพื้นที่เสมือนและแปล โดยการจับคู่ตำแหน่งที่อยู่เป็นที่อยู่จริงเมื่อเข้าถึงหน่วยความจำ
- word หน่วยปกติของการเข้าถึงในคอมพิวเตอร์โดย ปกติจะเป็นกลุ่ม 32 บิต
- wafer แผ่นเวเฟอร์แผ่นจากแท่งซิลิกอนหนาไม่เกิน 0.1 นิ้วใช้ทำเป็นแผ่น
- Yield ผลผลิตคิดเป็นเปอร์เซ็นต์ของแม่พิมพ์ที่ดี จากจำนวนทั้งหมดของแม่พิมพ์ บนแผ่นเวเฟอร์

ในสถาปัตยกรรม RISC คำสั่งมักถูกประมวลผลในไปป์ไลน์ที่มีหลายขั้นตอน เมื่อมีการกระโดด (branch เช่น beq) CPU ต้องเปลี่ยนที่อยู่ของคำสั่งถัดไป (program counter หรือ PC) ซึ่งอาจทำให้ไปป์ไลน์หยุดชะงัก (pipeline stall) เพราะต้องรอคำสั่งใหม่จาก

ที่อยู่เป้าหมาย ในระหว่างที่ CPU ตัดสินใจว่าจะกระโดดหรือไม่ (เช่น ในคำสั่ง beq ต้องตรวจสอบเงื่อนไข) คำสั่งถัดจากคำสั่งกระโดด (ที่อยู่ PC+4) จะถูกดึง (fetch) เข้ามาในไปป์ไลน์แล้ว หากกระโดดเกิดขึ้น คำสั่งที่ดึงมานี้จะไม่ถูกใช้ และต้องเริ่มดึงคำสั่งจากที่อยู่ใหม่ ทำให้เสียรอบการทำงาน (cycle) คอมไพเลอร์หรือนักเขียนแอสเซมบลีต้องเลือกคำสั่งที่เหมาะสมสำหรับ delay slot เพื่อให้มีประโยชน์และไม่ก่อให้เกิดผลข้างเคียงที่ไม่ต้องการ เช่น คำสั่งที่ไม่ขึ้นกับผลของการกระโดด (เช่น การคำนวณที่ต้องทำอยู่แล้ว) หรือคำสั่งที่เป็นส่วนหนึ่งของรอบก่อนหน้า หากไม่มีคำสั่งที่เหมาะสม คอมไพเลอร์จะใส่ nop (no operation) ใน delay slot ซึ่งเสียประโยชน์จากรอบนั้น RISC-V ออกแบบให้เรียบง่ายและลดความซับซ้อนสำหรับคอมไพเลอร์ สถาปัตยกรรม RISC-V (เช่น RV32I, RV32V) ไม่ใช้ delay slot เพราะออกแบบให้เรียบง่ายและพึ่งพาเทคโนโลยีไปป์ไลน์ที่ทันสมัย เช่น branch prediction

C.9 ข้อสรุปท้ายบท

RISC-V เป็น ISA แบบเปิดที่ปฏิวัติวงการด้วยความเรียบง่าย ออกแบบให้เป็นโมดูลาร์ มีชุดคำสั่งพื้นฐาน (เช่น RV32I) และส่วนขยาย (เช่น RV32V) ผู้ใช้สามารถเลือกส่วนขยายตามความต้องการ RV32I มีคำสั่งพื้นฐานสำหรับจำนวนเต็ม 32 บิต มี 47 คำสั่งและ 32 รีจิสเตอร์ ทำให้ง่ายต่อการเรียนรู้ และ RV64 ขยายเป็น 64 บิต รองรับคำสั่งเดียวกันแต่จัดการข้อมูลขนาดใหญ่กว่า ส่วนขยาย RV32M เพิ่มคำสั่งคูณและหาร RV32A รองรับการทำงานแบบอะตอมิก RV32V รองรับงานประมวลผลสูง เช่น การคำนวณเวกเตอร์ใน DAXPY ความยืดหยุ่น และการไม่มีค่าใบอนุญาต จึงเหมาะสำหรับงานหลากหลาย ตั้งแต่ IoT (เช่น ESP32-C3) ไปจนถึงการคำนวณประสิทธิภาพสูง ชิป SiFive Performance P550 ใช้ในเซิร์ฟเวอร์ ชิป Kendryte K210 (RISC-V) รองรับการประมวลผลภาพและ ML มีคำสั่งและส่วนขยายที่สามารถกำหนดเองตอบโจทย์งานเฉพาะ RV32E เป็นส่วนขยายของ RISC-V ที่ออกแบบสำหรับระบบฝังตัว (embedded systems) โดยลดจำนวนรีจิสเตอร์จาก 32 ตัวใน RV32I เหลือ 16 ตัว (x0 ถึง x15) เพื่อลดต้นทุนและขนาดของฮาร์ดแวร์ นอกจากนี้ RISC-V มีส่วนขยายที่กำหนดเอง ผู้ผลิตสามารถเพิ่มคำสั่งเฉพาะ เช่น ในชิป AI หรือ IoT การพัฒนา RISC-V สามารถทำได้บนแพลตฟอร์มอย่าง Raspberry Pi โดยใช้ JTAG และเครื่องมือเช่น OpenOCD นอกจากนี้ RISC-V ยังมีชุมชนที่กำลังเติบโต โดยได้รับการสนับสนุนจาก RISC-V International และบริษัทชั้นนำ RISC-V มีระบบปฏิบัติการที่รองรับ RISC-V ในปี 2025 มีหลากหลาย ตั้งแต่ระบบปฏิบัติการทั่วไป เช่น Linux ไปจนถึง RTOS สำหรับระบบฝังตัว การสนับสนุนที่กว้างขวางนี้แสดงถึงความยืดหยุ่นของ RISC-V ในระบบฝังตัว, IoT, และการประมวลผลทั่วไป ทำให้ RISC-V เหมาะสำหรับการเรียนรู้ ISA และภาษาแอสเซมบลี

ดัชนี

ABI, *ดูเพิ่มเติม* application binary interface
Add, 22, *165*
  immediate, 22, *165*
  immediate word, 124, *165*
  upper immediate to PC, *171*
  word, 124, *165*
add, 22, 93, *ดูเพิ่มเติม* c.add, *165*
Add upper immediate to PC, 22
add.uw, 128
addi, *ดูเพิ่มเติม* Add immediate, *ดูเพิ่มเติม* c.addi16sp, *ดูเพิ่มเติม* c.addi4spn, *ดูเพิ่มเติม* c.addi, *ดูเพิ่มเติม* c.li, *165*
addiw, *ดูเพิ่มเติม* Add immediate word, *ดูเพิ่มเติม* c.addiw, *165*
addw, *ดูเพิ่มเติม* Add word, *ดูเพิ่มเติม* c.addw, *165*
ALGOL, *162*
Allen, Fran, *16*
AMD64, 132
amoadd.d, *ดูเพิ่มเติม* Atomic Memory Operation Add Doubleword, *166*
amoadd.w, *ดูเพิ่มเติม* Atomic Memory Operation Add Word, *166*
amoand.d, *ดูเพิ่มเติม* Atomic Memory Operation And Doubleword, *166*
amoand.w, *ดูเพิ่มเติม* Atomic Memory Operation And Word, *166*
amomax.d, *ดูเพิ่มเติม* Atomic Memory Operation Maximum Doubleword, *167*
amomax.w, *ดูเพิ่มเติม* Atomic Memory Operation Maximum Word, *167*
amomaxu.d, *ดูเพิ่มเติม* Atomic Memory Operation Maximum Unsigned Doubleword, *167*
amomaxu.w, *ดูเพิ่มเติม* Atomic Memory Operation Maximum Unsigned Word, *167*
amomin.d, *ดูเพิ่มเติม* Atomic Memory Operation Minimum Doubleword, *168*
amomin.w, *ดูเพิ่มเติม* Atomic Memory Operation Minimum Word, *168*
amominu.d, *ดูเพิ่มเติม* Atomic Memory Operation Minimum Unsigned Doubleword, *168*
amominu.w, *ดูเพิ่มเติม* Atomic Memory Operation Minimum Unsigned Word, *168*
amoor.d, *ดูเพิ่มเติม* Atomic Memory Operation Or Doubleword, *169*
amoor.w, *ดูเพิ่มเติม* Atomic Memory Operation Or Word, *169*
amoswap.d, *ดูเพิ่มเติม* Atomic Memory Operation Swap Doubleword, *169*
amoswap.w, *ดูเพิ่มเติม* Atomic Memory Operation Swap Word, *169*
amoxor.d, *ดูเพิ่มเติม* Atomic Memory Operation Exclusive Or Doubleword, *170*
amoxor.w, *ดูเพิ่มเติม* Atomic Memory Operation Exclusive Or Word, *170*
And, 22, *170*
  immediate, 22, *170*
and, *ดูเพิ่มเติม* c.and, *170*
andi, 22, *ดูเพิ่มเติม* And immediate, *ดูเพิ่มเติม* c.andi, *170*
andn, **121**
application binary interface, 20, 31, 37, 55
Application Specific Integrated Circuits, *2*
architecture, *8*
ARM
  code size, 10, 132
  Cortex-A5, 7
  Cortex-A9, 8
  DAXPY, **60**
  Insertion Sort, **28**
  instruction reference manual
    number of pages, 14
  Load Multiple, 7, 9
  number of registers, 11
  Thumb, 9, 10
  Thumb-2, 9, 10
  Tree Sum, **222**
ARM SVE, 109
ARM64 SVE Vector, *108*
ARMv8, **132**
Art is long, life is short, *228*
ASIC, *ดูเพิ่มเติม* Application Specific Integrated Circuits
assembler directives, 43
Atomic Memory Operation
  Add
    Doubleword, 124, *166*
    Word, 68, *166*
  And
    Doubleword, 124, *166*
    Word, 68, *166*
  Exclusive Or
    Doubleword, 124, *170*
    Word, 68, *170*
  Maximum
    Doubleword, 124, *167*
    Word, 68, *167*
  Maximum Unsigned
    Doubleword, 124, *167*
    Word, 68, *167*
  Minimum
    Doubleword, 124, *168*

Word, 68, *168*
Minimum Unsigned
Doubleword, 124, *168*
Word, 68, *168*
Or
Doubleword, 124, *169*
Word, 68, *169*
Swap
Doubleword, 124, *169*
Word, 68, *169*
auipc, *ดูเพิ่มเติม* Add upper immediate to PC, *171*

backwards
binary-compatibility, *4*
bclr, **122**
Bell, C. Gordon, 52, *124*
beq, *ดูเพิ่มเติม* Branch if equal, *ดูเพิ่มเติม* c.beqz, *171*
beqz, *39*, *171*
bext, **122**
bge, *ดูเพิ่มเติม* Branch if greater or equal, *171*
bgeu, *ดูเพิ่มเติม* Branch if greater or equal unsigned, *171*
bgez, *39*, *172*
bgt, *39*, *172*
bgtu, *39*, *172*
bgtz, *39*, *172*
Bhirasri, Silpa, *228*
binv, **122**
ble, *39*, *172*
bleu, *39*, *172*
blez, *39*, *172*
blt, *ดูเพิ่มเติม* Branch if less than, *173*
bltu, *ดูเพิ่มเติม* Branch if less than unsigned, *173*
bltz, *39*, *173*
bne, *ดูเพิ่มเติม* Branch if not equal, *ดูเพิ่มเติม* c.bnez, *173*
bnez, *39*, *173*
Boole, George , *120*
Branch
if equal, 25, *171*
if greater or equal, 25, *171*
if greater or equal unsigned, 25, *171*
if less than, 25, *173*
if less than unsigned, 25, *173*
if not equal, 25, *173*
branch prediction, 9, 22
Brooks, Fred, 158
Browning, Robert, *61*
bset, **122**

C programs
DAXPY, **60**
Insertion Sort, **28**
Tree Sum, **222**
c.add, *ดูเพิ่มเติม* add, *173*
c.addi, *ดูเพิ่มเติม* addi, *174*
c.addi16sp, *ดูเพิ่มเติม* addi, *174*
c.addi4spn, *ดูเพิ่มเติม* addi, *174*
c.addiw, 124, *ดูเพิ่มเติม* addiw, *174*
c.addw, 124, *ดูเพิ่มเติม* addw, *174*
c.and, *ดูเพิ่มเติม* and, *174*
c.andi, *ดูเพิ่มเติม* andi, *175*
c.beqz, *ดูเพิ่มเติม* beq, *175*
c.bnez, *ดูเพิ่มเติม* bne, *175*
c.ebreak, *ดูเพิ่มเติม* ebreak, *175*
c.fld, *ดูเพิ่มเติม* fld, *175*
c.fldsp, *ดูเพิ่มเติม* fld, *175*
c.flw, *ดูเพิ่มเติม* flw, *176*
c.flwsp, *ดูเพิ่มเติม* flw, *176*
c.fsd, *ดูเพิ่มเติม* fsd, *176*
c.fsdsp, *ดูเพิ่มเติม* fsd, *176*
c.fsw, *ดูเพิ่มเติม* fsw, *176*
c.fswsp, *ดูเพิ่มเติม* fsw, *177*
c.j, *ดูเพิ่มเติม* jal, *177*
c.jal, *ดูเพิ่มเติม* jal, *177*
c.jalr, *ดูเพิ่มเติม* jalr, *177*
c.jr, *ดูเพิ่มเติม* jalr, *177*
c.ld, 124, *ดูเพิ่มเติม* ld, *177*
c.ldsp, 124, *ดูเพิ่มเติม* ld, *178*
c.li, *ดูเพิ่มเติม* addi, *178*
c.lui, *ดูเพิ่มเติม* lui, *178*
c.lw, *ดูเพิ่มเติม* lw, *178*
c.lwsp, *ดูเพิ่มเติม* lw, *178*
c.mv, *ดูเพิ่มเติม* add, *178*
c.or, *ดูเพิ่มเติม* or, *179*
c.sd, 124, *ดูเพิ่มเติม* sd, *179*
c.sdsp, 124, *ดูเพิ่มเติม* sd, *179*
c.slli, *ดูเพิ่มเติม* slli, *179*
c.srai, *ดูเพิ่มเติม* srai, *179*
c.srli, *ดูเพิ่มเติม* srli, *179*
c.sub, *ดูเพิ่มเติม* sub, *180*
c.subw, 124, *ดูเพิ่มเติม* subw, *180*
c.sw, *ดูเพิ่มเติม* sw, *180*
c.swsp, *ดูเพิ่มเติม* sw, *180*
c.xor, *ดูเพิ่มเติม* xor, *180*
call, *39*, *180*
Callee saved registers, *37*
Caller saved registers, *37*
Calling conventions, **36**
Carryless multiply, **122**
Chanel, Coco, 164
chip, *ดูเพิ่มเติม* die, *7*
clmul, **122**
clmulh, **122**
clmulr, **122**
clz, **121**
clzw, 128
Compilers
Turing Award, *162*
Control and Status Register
`marchid`, *158*, 158
`mcause`, *144*, 146, 147
`mcounteren`, *158*, 158
`mcycle`, *158*, 158
`medeleg`, *151*, 152
`mepc`, *144*, 146, 147
`mhartid`, *158*, 158
`mhpmcounter`*i*, *158*, 158
`mhpmevent`*i*, *158*, 158
`mideleg`, *151*
`mie`, *144*, 146, 149
`mimpid`, *158*, 158
`minstret`, *158*, 158
`mip`, *144*, 146, 149
`misa`, 157, *158*
`mscratch`, *144*, 146, 147
`mstatus`, *144*, 146, 147, 149
`mtimecmp`, *158*, 158
`mtime`, *158*, 158
`mtval`, *144*, 146, 147
`mtvec`, *144*, 146, 147
`mvendorid`, 157, *158*
`pmpcfg`, *150*
`satp`, *155*, 156
`scause`, *144*, 153
`scounteren`, *158*, 158
`sedeleg`, *151*
`sepc`, *144*, 153
`sideleg`, *151*
`sie`, *151*, 152
`sip`, *151*, 152
`sscratch`, *144*, 153
`sstatus`, *152*, 153
`stval`, *144*, 153
`stvec`, *144*, 153
read and clear, 27, *181*
read and clear immediate, 27, *181*
read and set, 27, *182*
read and set immediate, 27, *182*
read and write, 27, *182*
read and write immediate, 27, *182*
CoreMark benchmark, 8
cost, *ดูเพิ่มเติม* instruction set architecture, principles of design, cost
cpopw, 128
Cray, Seymour, *80*, 133
CRC32, **122**

CSR, *ดู* Control and Status Register
csrc, *39*, *181*
csrci, *39*, *181*
csrr, *39*, *181*
csrrc, *ดูเพิ่มเติม* Control and Status Register read and clear, *181*
csrrci, *ดูเพิ่มเติม* Control and Status Register read and clear immediate, *181*
csrrs, *ดูเพิ่มเติม* Control and Status Register read and set, *182*
csrrsi, *ดูเพิ่มเติม* Control and Status Register read and set immediate, *182*
csrrw, *ดูเพิ่มเติม* Control and Status Register read and write, *182*
csrrwi, *ดูเพิ่มเติม* Control and Status Register read and write immediate, *182*
csrs, *39*, *183*
csrsi, *39*, *183*
csrw, *39*, *183*
csrwi, *39*, *183*
ctzw, 128

da Vinci, Leonardo, *2*
data-level parallelism, *80*
DAXPY, **60**, 91, 105, 109, 112
de Saint-Exupéry, Antoine, *54*
delay slot, 9
delayed branch, 9
die, *ดูเพิ่มเติม* chip, *7*
  yield, *7*
div, *183*
Divide, 51, *183*
  unsigned, 51, *184*
  unsigned word, 124, *184*
  using shift right, 51
  word, 124, *184*
divu, *ดูเพิ่มเติม* Divide unsigned, *184*
divuw, *ดูเพิ่มเติม* Divide unsigned word, *184*
divw, *ดูเพิ่มเติม* Divide word, *184*
dynamic linking, **47**
dynamic typing, *85*, 130

ease of programming, compiling, and linking, *ดูเพิ่มเติม* instruction set architecture, principles of design, ease of programming, compiling, and linking
ebreak, *184*
ecall, *184*
EEW, *86*, 93
Effective Element Width, *86*
Effective register length multipliler, *86*
Einstein, Albert, *68*
Element length, *86*
ELEN, *86*
ELF, *ดูเพิ่มเติม* executable and linkable format
EMUL, *86*
endianness, 24
epilogue, *ดูเพิ่มเติม* function epilogue
Exception, *143*
Exception Return
  Machine, *147*, *207*
  Supervisor, *153*, *217*
Exclusive Or, 22, *219*
  immediate, 22, *219*
executable and linkable format, *43*

fabs.d, *39*, *185*
fabs.s, *39*, *185*
fadd.d, *ดูเพิ่มเติม* Floating-point Add double-precision, *185*
fadd.s, *ดูเพิ่มเติม* Floating-point Add single-precision, *185*
fclass.d, *ดูเพิ่มเติม* Floating-point Classify double-precision, *185*
fclass.s, *ดูเพิ่มเติม* Floating-point Classify single-precision, *186*
fcvt.d.l, *ดูเพิ่มเติม* Floating-point Convert double from long, *186*
fcvt.d.lu, *ดูเพิ่มเติม* Floating-point Convert double from long unsigned, *186*
fcvt.d.s, *ดูเพิ่มเติม* Floating-point Convert double from single, *187*
fcvt.d.w, *ดูเพิ่มเติม* Floating-point Convert double from word, *187*
fcvt.d.wu, *ดูเพิ่มเติม* Floating-point Convert double from word unsigned, *187*
fcvt.l.d, *ดูเพิ่มเติม* Floating-point Convert long from double, *187*
fcvt.l.s, *ดูเพิ่มเติม* Floating-point Convert long from single, *188*
fcvt.lu.d, *ดูเพิ่มเติม* Floating-point Convert long unsigned from double, *188*
fcvt.lu.s, *ดูเพิ่มเติม* Floating-point Convert long unsigned from single, *188*
fcvt.s.d, *ดูเพิ่มเติม* Floating-point Convert single from double, *188*
fcvt.s.l, *ดูเพิ่มเติม* Floating-point Convert single from long, *189*
fcvt.s.lu, *ดูเพิ่มเติม* Floating-point Convert single from long unsigned, *189*
fcvt.s.w, *189*
fcvt.s.wu, *ดูเพิ่มเติม* Floating-point Convert single from word unsigned, *189*
fcvt.w.d, *ดูเพิ่มเติม* Floating-point Convert word from double, *190*
fcvt.w.s, *ดูเพิ่มเติม* Floating-point Convert word from single, *190*
fcvt.wu.d, *ดูเพิ่มเติม* Floating-point Convert word unsigned from double, *190*
fcvt.wu.s, *ดูเพิ่มเติม* Floating-point Convert word unsigned from single, *190*
fdiv.d, *ดูเพิ่มเติม* Floating-point Divide double-precision, *191*
fdiv.s, *ดูเพิ่มเติม* Floating-point Divide single-precision, *191*
Fence
  Instruction Stream, *191*
  Memory and I/O, *191*
  Virtual Memory, *157*, *213*
fence, *39*, *ดูเพิ่มเติม* Fence Memory and I/O, *191*
fence.i, *ดูเพิ่มเติม* Fence Instruction Stream, *191*
feq.d, *ดูเพิ่มเติม* Floating-point Equals double-precision, *192*
feq.s, *ดูเพิ่มเติม* Floating-point Equals single-precision, *192*
Field-Programmable Gate Array, *2*
find-first-set mask bit, *99*
fld, *ดูเพิ่มเติม* c.fldsp, *ดูเพิ่มเติม* c.fld, *ดูเพิ่มเติม* Floating-point load doubleword, *192*
fle.d, *ดูเพิ่มเติม* Floating-point Less or Equals double-precision, *192*, *ดูเพิ่มเติม* Floating-point Less Than double-precision
fle.s, *ดูเพิ่มเติม* Floating-point Less or Equals single-precision, *ดูเพิ่มเติม*

Floating-point Less Than
single-precision, *193*
Floating-Point
dynamic rounding mode, *57*
fused multiply-add, *58*
IEEE 754-2019 floating-point
standard, 54
sign-injection, *59*
static rounding mode, *57*
Floating-point, **54**
Add
double-precision, 54, *185*
half-precision, 54
single-precision, 54, *185*
binary128, **61**
binary16, **61**
binary32, **61**
binary64, **61**
Classify
double-precision, 54, *185*
half-precision, 54
single-precision, 54, *186*
Convert
double from half, 54
double from long, 124,
186
double from long
unsigned, 124, *186*
double from single, 54,
187
double from word, 54,
187
double from word
unsigned, 54, *187*
half from double, 54
half from long, 124
half from long unsigned,
124
half from single, 54
half from word, 54
half from word unsigned,
54
long from double, 124,
187
long from half, 124
long from single, 124, *188*
long unsigned from
double, 124, *188*
long unsigned from half,
124
long unsigned from
single, 124, *188*
single from double, 54,
188
single from half, 54
single from long, 124, *189*
single from long
unsigned, 124, *189*
single from word, 54
single from word
unsigned, 54, *189*
word from double, 54,
190
word from half, 54
word from single, 54, *190*
word unsigned from
double, 54, *190*
word unsigned from half,
54
word unsigned from
single, 54, *190*
decimal128, **61**
decimal32, **61**
decimal64, **61**
Divide
double-precision, 54, *191*
half-precision, 54
single-precision, 54, *191*
Equals
double-precision, 54, *192*
half-precision, 54
single-precision, 54, *192*
Fused multiply-add
double-precision, 54, *194*
single-precision, 54, *194*
Fused multiply-half
single-precision, 54
Fused multiply-subtract
double-precision, 54, *195*
half-precision, 54
single-precision, 54, *195*
Fused negative multiply-add
double-precision, 54, *198*
half-precision, 54
single-precision, 54, *198*
Fused negative
multiply-subtract
double-precision, 54, *198*
half-precision, 54
single-precision, 54, *199*
half precision, **61**
Less or Equals
double-precision, 54, *192*
half-precision, 54
single-precision, 54, *193*
Less Than
double-precision, 54, *193*
half-precision, 54
single-precision, 54, *193*
Load
doubleword, 54, *192*
half, 54
word, 54, *193*
Maximum
double-precision, 54, *194*
half-precision, 54
single-precision, 54, *194*
Minimum
double-precision, 54, *195*
half-precision, 54
single-precision, 54, *195*
Move
doubleword from integer,
196
doubleword to integer,
197
half from integer, 54
half to integer, 54
single from integer, 54
single to integer, 54
word from integer, *197*
word to integer, *197*
Multiply
double-precision, 54, *196*
half-precision, 54
single-precision, 54, *196*
octuple precision, **61**
quadruple precision, **61**
Sign-inject
double-precision, 54, *200*
half-precision, 54
single-precision, 54, *200*
Sign-inject negative
double-precision, 54, *201*
half-precision, 54
single-precision, 54, *201*
Sign-inject XOR
double-precision, 54, *201*
half-precision, 54
single-precision, 54, *201*
Square root
double-precision, 54, *202*
half-precision, 54
single-precision, 54, *202*
Store
doubleword, 54, *200*
half, 54
word, 54, *203*
Subtract
double-precision, 54, *202*
half-precision, 54
single-precision, 54, *203*
floating-point
control and status register,
55
flt.d, *193*
flt.s, *193*
flw, *ดูเพิ่มเติม* c.flwsp, *ดูเพิ่มเติม*
c.flw, *ดูเพิ่มเติม* Floating-point
load word, *193*

fmadd.d, *ดูเพิ่มเติม* Floating-point fused multiply-add double-precision, *194*
fmadd.s, *ดูเพิ่มเติม* Floating-point fused multiply-add single-precision, *194*
fmax.d, *ดูเพิ่มเติม* Floating-point maximum double-precision, *ดูเพิ่มเติม* Floating-point maximum single-precision, *194*
fmax.s, *194*
fmin.d, *ดูเพิ่มเติม* Floating-point minimum double-precision, *195*
fmin.s, *ดูเพิ่มเติม* Floating-point minimum single-precision, *195*
fmsub.d, *ดูเพิ่มเติม* Floating-point fused multiply-subtract double-precision, *195*
fmsub.s, *ดูเพิ่มเติม* Floating-point fused multiply-subtract single-precision, *195*
fmul.d, *ดูเพิ่มเติม* Floating-point Multiply double-precision, *196*
fmul.s, *ดูเพิ่มเติม* Floating-point Multiply single-precision, *196*, *ดูเพิ่มเติม* Floating-point Subtract single-precision
fmv.d, *39*, *196*
fmv.d.x, *ดูเพิ่มเติม* Floating-point move doubleword from integer, *196*
fmv.s, *39*, *196*
fmv.w.x, *ดูเพิ่มเติม* Floating-point move word from integer, *197*
fmv.x.d, *ดูเพิ่มเติม* Floating-point move doubleword to integer, *197*
fmv.x.w, *ดูเพิ่มเติม* Floating-point move word to integer, *197*
fneg.d, *39*, *197*
fneg.s, *39*, *197*
fnmadd.d, *ดูเพิ่มเติม* Floating-point fused negative multiply-add double-precision, *ดูเพิ่มเติม* Floating-point fused negative multiply-add single-precision, *198*
fnmadd.s, *198*
fnmsub.d, *ดูเพิ่มเติม* Floating-point fused negative multiply-subtract double-precision, *198*
fnmsub.s, *ดูเพิ่มเติม* Floating-point fused negative multiply-subtract single-precision, *199*
FPGA, *ดูเพิ่มเติม* Field-Programmable Gate Array, *2*
frcsr, *39*, *199*
frflags, *39*, *199*
frrm, *39*, *199*
fscsr, *39*, *199*
fsd, *ดูเพิ่มเติม* c.fsdsp, *ดูเพิ่มเติม* c.fsd, *ดูเพิ่มเติม* Floating-point store doubleword, *200*
fsflags, *39*, *200*
fsgnj.d, *ดูเพิ่มเติม* Floating-point Sign-inject double-precision, *200*
fsgnj.s, *ดูเพิ่มเติม* Floating-point Sign-inject single-precision, *200*
fsgnjn.d, *ดูเพิ่มเติม* Floating-point Sign-inject negative double-precision, *201*
fsgnjn.s, *ดูเพิ่มเติม* Floating-point Sign-inject negative single-precision, *201*
fsgnjx.d, *ดูเพิ่มเติม* Floating-point Sign-inject XOR double-precision, *201*
fsgnjx.s, *ดูเพิ่มเติม* Floating-point Sign-inject XOR single-precision, *201*
fsqrt.d, *ดูเพิ่มเติม* Floating-point Square Root double-precision, *202*
fsqrt.s, *ดูเพิ่มเติม* Floating-point Square Root single-precision, *202*
fsrm, *39*, *202*
fsub.d, *ดูเพิ่มเติม* Floating-point Subtract double-precision, *202*
fsub.s, *203*
fsw, *ดูเพิ่มเติม* c.fswsp, *ดูเพิ่มเติม* c.fsw, *ดูเพิ่มเติม* Floating-point store word, *203*
function epilogue, *39*
function prologue, *39*
Fused multiply-add, *58*

gather, *88*

Hart, *141*

IEEE 754-2008, 61
IEEE 754-2019, 61
IEEE 754-2019 floating-point standard, 54
Illiac IV, *111*
implementation, *8*
Insertion Sort, **28**
Instruction diagram
  Privileged instructions, **141**
  RV32A, **68**
  RV32B, **121**
  RV32C, **72**
  RV32D, **54**
  RV32F, **54**
  RV32I, **16**
  RV32M, **50**
  RV32V, **81**
  RV64A, **124**
  RV64B, **128**
  RV64C, **124**
  RV64D, **124**
  RV64F, **124**
  RV64I, **124**
  RV64M, **124**
  Zfh, 54
instruction set architecture, *3*
  backwards binary-compatibility, *4*
  elegance, 14, 29, 48, 75, 113, 133, 163
  incremental, **4**
  metrics of design, **6**
    cost, **6**, 16, 19, 23, 24, 29, 52, 74, 122, 132, 157
    ease of programming, compiling, and linking, **11**, 19, 20, *22*, 24, 25, 29, 59, 80, 85, 88, 89, 112, 130, 133, 147, 158
    isolation of architecture from implementation, **8**, 29, 80, 112, 141
    performance, **8**, 16, 20, 29, 51, 54, 57, 61, 70, 80, 89, 93, 102, 110, 130, 132
    program size, **9**, 29, 72, 75, 128, 132, 133
    room for growth, **9**, 19, 29, 133
    simplicity, **7**, 13, 14, 20, 22--27, *29*, 61, 70, 72, 81, 109, 147, 153, 158
  mistakes of the past, **29**
  modularity, **5**

open, **3**
principles of design
cost, 48
ease of programming,
compiling, and linking, 45, 47
performance, 37, 48, 162
room for growth, 163
simplicity, 39, 43
Interrupt, *144*
ISA, *ดู* instruction set architecture
isolation of architecture from implementation, *ดูเพิ่มเติม* instruction set architecture, principles of design, isolation of architecture from implementation
Itanium, **131**

j, *39*, *203*
jal, *39*, *ดูเพิ่มเติม* c.jal, *ดูเพิ่มเติม* c.j, *ดูเพิ่มเติม* Jump and link, *203*
jalr, *39*, *ดูเพิ่มเติม* c.jalr, *ดูเพิ่มเติม* c.jr, *ดูเพิ่มเติม* Jump and link register, *204*
Johnson, Kelly, *47*
jr, *39*, *204*
Jump and link, 26, *203*
register, 26, *204*

Kuck, David, *111*

la, *204*
lb, *ดูเพิ่มเติม* Load byte, *204*
lbu, *ดูเพิ่มเติม* Load byte unsigned, *204*
ld, *ดูเพิ่มเติม* c.ldsp, *ดูเพิ่มเติม* c.ld, *ดูเพิ่มเติม* Load doubleword, *205*
leaf function, *37*
lh, *ดูเพิ่มเติม* Load halfword, *205*
lhu, *ดูเพิ่มเติม* Load halfword unsigned, *205*
li, *39*, *205*
Lindy effect, *29*
linker relaxation, **46**
little-endian, 24
lla, *205*
LMUL, *86*, 86, 92, 95, 111
Load
byte, 24, *204*
byte unsigned, 24, *204*
doubleword, 124, *205*
halfword, 24, *205*
halfword unsigned, 24, *205*
reserved
doubleword, 124, *206*
word, 68, *206*
upper immediate, 22, *206*
word, 24, *206*
word unsigned, 24, *207*
Load upper immediate, 22
lr.d, *ดูเพิ่มเติม* Load reserved doubleword, *206*
lr.w, *ดูเพิ่มเติม* Load reserved word, *206*
lui, *ดูเพิ่มเติม* c.lui, *ดูเพิ่มเติม* Load upper immediate, *206*
lw, *ดูเพิ่มเติม* c.lwsp, *ดูเพิ่มเติม* c.lw, *ดูเพิ่มเติม* Load word, *206*
lwu, *ดูเพิ่มเติม* Load word unsigned, *207*

Machine mode, *141*
macrofusion, *7*, 7, 74
marchid, *ดู* Control and Status Register
max, **121**
maximum vector length, *85*
maxu, **121**
mcause, *ดู* Control and Status Register
mcounteren, *ดู* Control and Status Register
mcycle, *ดู* Control and Status Register
mepc, *ดู* Control and Status Register
metrics of ISA design, *ดูเพิ่มเติม* instruction set architecture, metrics of design
mhartid, *ดู* Control and Status Register
mhpmcounter*i*, *ดู* Control and Status Register
mhpmevent*i*, *ดู* Control and Status Register
microMIPS, 60
mie, *ดู* Control and Status Register
mimpid, *ดู* Control and Status Register
min, **121**
minstret, *ดู* Control and Status Register
minu, **121**
mip, *ดู* Control and Status Register
MIPS
assembler, 23
DAXPY, **60**
delayed branch, 9, 25, 31, 62, 136
delayed load, 24, 31, 136
Insertion Sort, **28**
MIPS-IV, **132**
misa, *ดู* Control and Status Register
Moore's Law, 2
mret, *ดูเพิ่มเติม* Exception Return Machine, *207*
mscratch, *ดู* Control and Status Register
mstatus, *ดู* Control and Status Register
mtime, *ดู* Control and Status Register
mtimecmp, *ดู* Control and Status Register
mtval, *ดู* Control and Status Register
mtvec, *ดู* Control and Status Register
mul, *ดูเพิ่มเติม* Multiply, *207*
mulh, *ดูเพิ่มเติม* Multiply high, *207*
mulhsu, *ดูเพิ่มเติม* Multiply high signed-unsigned, *208*
mulhu, *ดูเพิ่มเติม* Multiply high unsigned, *208*
Multiply, 51, *207*
high, 51, *207*
high signed-unsigned, 51, *208*
high unsigned, 51, *208*
multi-word, 52
using shift left, 51
word, 124, *208*
mulw, *ดูเพิ่มเติม* Multiply word, *208*
mv, *39*, *208*
mvendorid, *ดู* Control and Status Register

Narrowing instructions, *93*, 94
neg, *39*, *208*
negw, *39*, *208*
nop, *39*, *209*
not, *39*, *209*

Occam, William of, *50*
Or, 22, *209*
immediate, 22, *209*
or, *ดูเพิ่มเติม* c.or, *209*
ori, 22, *ดูเพิ่มเติม* Or immediate, *209*
orn, **121**
out-of-order processors, *20*

Page, *154*
Page fault, *154*

Page table, *154*
Pascal, Blaise, 75
performance, *ดูเพิ่มเติม* instruction set architecture, principles of design, performance
  CoreMark benchmark, 8
  equation, 8
Perlis, Alan, *162*, 162
PIC, *ดูเพิ่มเติม* position independent code
pipelined processor, *22*
Plato, *220*
popc, **121**
position independent code, 11, 25, *45*, 131
Privilege mode, *140*
  Machine mode, *141*
  User mode, *149*
program size, *ดูเพิ่มเติม* instruction set architecture, principles of design, program size
Programming languages
  Turing Award, *162*
prologue, *ดูเพิ่มเติม* function prologue
Pseudoinstruction, **39**
  beqz, *39*, *171*
  bgez, *39*, *172*
  bgt, *39*, *172*
  bgtu, *39*, *172*
  bgtz, *39*, *172*
  ble, *39*, *172*
  bleu, *39*, *172*
  blez, *39*, *172*
  bltz, *39*, *173*
  bnez, *39*, *173*
  call, *39*, *180*
  csrc, *39*, *181*
  csrci, *39*, *181*
  csrr, *39*, *181*
  csrs, *39*, *183*
  csrsi, *39*, *183*
  csrw, *39*, *183*
  csrwi, *39*, *183*
  fabs.d, *39*, *185*
  fabs.s, *39*, *185*
  fence, *39*
  fmv.d, *39*, *196*
  fmv.s, *39*, *196*
  fneg.d, *39*, *197*
  fneg.s, *39*, *197*
  frcsr, *39*, *199*
  frflags, *39*, *199*
  frrm, *39*, *199*
  fscsr, *39*, *199*
  fsflags, *39*, *200*
  fsrm, *39*, *202*
  j, *39*, *203*
  jr, *39*, *204*
  la, *204*
  li, *39*, *205*
  lla, *205*
  mv, *39*, *208*
  neg, *39*, *208*
  negw, *39*, *208*
  nop, *39*, *209*
  not, *39*, *209*
  rdcycle, *39*, *209*
  rdcycleh, *39*, *209*
  rdinstret, *39*, *210*
  rdinstreth, *39*, *210*
  rdtime, *39*, *210*
  rdtimeh, *39*, *210*
  ret, *39*, *211*
  seqz, *39*, *212*
  sext.w, *39*, *212*
  sgtz, *39*, *213*
  sltz, *39*, *215*
  snez, *39*, *215*
  tail, *39*, *219*

Ralph Brazelton Peck, 122
rdcycle, *39*, *209*
rdcycleh, *39*, *209*
rdinstret, *39*, *210*
rdinstreth, *39*, *210*
rdtime, *39*, *210*
rdtimeh, *39*, *210*
reductions, *99*
registers
  number of, 11
rem, *ดูเพิ่มเติม* Remainder, *210*
Remainder, 51, *210*
  unsigned, 51, *210*
  unsigned word, 124, *211*
  word, 124, *211*
remu, *ดูเพิ่มเติม* Remainder unsigned, *210*
remuw, *ดูเพิ่มเติม* Remainder unsigned word, *211*
remw, *ดูเพิ่มเติม* Remainder word, *211*
ret, *39*, *211*
rev8, **122**
RISC-V
  application binary interface, 20, 31, 37, 55
  assembler directives, *43*
  BOOM, 8
  Calling conventions, **39**
  code size, 10, 132
  DAXPY, **60**
  function epilogue, *39*
  function prologue, *39*
  heap region, **45**
  Insertion Sort, **28**
  instruction reference manual
    number of pages, 14
  instruction set naming scheme, **5**
  International, **3**
  lessons learned, **29**
  Linker, **45**
  Loader, **47**
  long, *128*
  macrofusion, 7
  memory allocation, **45**
  modularity, *5*
  number of registers, 11
  pseudoinstruction, **39**
  pseudoinstructions, 12
  Rocket, 7
  RV128, *134*
  RV32A, **68**
  RV32B, **120**
  RV32C, 10, 12, **72**
  RV32D, **54**
  RV32F, **54**
  RV32G, *10*, 12
  RV32I, **16**
  RV32M, **50**
  RV32V, 12, **80**
  RV64A, **124**
  RV64C, **124**, **132**
  RV64D, **124**
  RV64F, **124**
  RV64G, 12
  RV64I, **124**
  RV64M, **124**
  saved registers, *37*
  stack region, **45**
  static region, **45**
  temporary registers, *37*
  text region, **45**
  Tree Sum, **222**
  Zfh, **54**
RISC-V ABI, *ดู* RISC-V Application Binary Interface, *46*
RISC-V Application Binary Interface
  ilp32, *46*
  ilp32d, *46*
  ilp32f, *46*
  lp64, *128*
  lp64d, *128*
  lp64f, *128*

RISC-V International, ii, **3**, 3, 5, 13, 29, 162, 163
rol, **121**
rolw, 128
room for growth, *ดูเพิ่มเติม* instruction set architecture, principles of design, room for growth
ror, **121**
roriw, 128
rorw, 128
RV128, *134*
RV32B, 162
RV32C, 60
RV32J, **162**
RV32P, **162**
RV32Q, **60**
RV64V, 109

Santayana, George, 28
sb, *ดูเพิ่มเติม* Store byte, *211*
sc.d, *ดูเพิ่มเติม* Store conditional doubleword, *211*
sc.w, *ดูเพิ่มเติม* Store conditional word, *212*
scatter, *88*
scause, *ดู* Control and Status Register
Schumacher, E. F., *72*
scounteren, *ดู* Control and Status Register
sd, *ดูเพิ่มเติม* c.sdsp, *ดูเพิ่มเติม* c.sd, *ดูเพิ่มเติม* Store doubleword, *212*
selected element width, *85*
sepc, *ดู* Control and Status Register
seqz, *39*, *212*
Set less than, *22*, *214*
  immediate, *22*, *215*
  immediate unsigned, *22*, *215*
  unsigned, *22*, *215*
set-before-first mask bit, *100*
set-including-first mask bit, *100*
set-only-first mask bit, *100*
seven metrics of ISA design, *ดูเพิ่มเติม* instruction set architecture, metrics of design
SEW, *85*, 86, *92*, *93*
sext, **121**
sext.w, *39*, *212*
sfence.vma, *ดูเพิ่มเติม* Fence Virtual Memory, *213*
sgtz, *39*, *213*
sh, *ดูเพิ่มเติม* Store halfword, *213*
sh1add, **121**
sh1add.uw, 128
sh2add, **121**
sh2add.uw, 128
sh3add, **121**
sh3add.uw, 128
Shift
  left logical, *22*, *213*
  left logical immediate, *22*, *214*
  left logical immediate word, 124, *214*
  left logical word, 124, *214*
  right arithmetic, *22*, *216*
  right arithmetic immediate, *22*, *216*
  right arithmetic immediate word, 124, *216*
  right arithmetic word, 124, *217*
  right logical, *22*, *217*
  right logical immediate, *22*, *217*
  right logical immediate word, 124, *218*
  right logical word, 124, *218*
SIMD, *ดูเพิ่มเติม* Single Instruction Multiple Data
simplicity, *ดูเพิ่มเติม* instruction set architecture, principles of design, simplicity
single bit manipulation, **122**
Single Instruction Multiple Data, 3, *12*
Single Instruction, Multiple Data, 80
sll, *ดูเพิ่มเติม* Shift left logical, *213*
slli, *ดูเพิ่มเติม* c.slli, *ดูเพิ่มเติม* Shift left logical immediate, *214*
slli.uw, 128
slliw, *ดูเพิ่มเติม* Shift left logical immediate word, *214*
sllw, *ดูเพิ่มเติม* Shift left logical word, *214*
slt, *ดูเพิ่มเติม* Set less than, *214*
slti, *ดูเพิ่มเติม* Set less than immediate, *215*
sltiu, *ดูเพิ่มเติม* Set less than immediate unsigned, *215*
sltu, *ดูเพิ่มเติม* Set less than unsigned, *215*
sltz, *39*, *215*
Small is Beautiful, *72*
Smith, Jim, *112*
snez, *39*, *215*
sra, *ดูเพิ่มเติม* Shift right arithmetic, *216*
srai, *ดูเพิ่มเติม* c.srai, *ดูเพิ่มเติม* Shift right arithmetic immediate, *216*
sraiw, *ดูเพิ่มเติม* Shift right arithmetic immediate word, *216*
sraw, *ดูเพิ่มเติม* Shift right arithmetic word, *217*
sret, *ดูเพิ่มเติม* Exception Return Supervisor, *217*
srl, *ดูเพิ่มเติม* Shift right logical, *217*
srli, *ดูเพิ่มเติม* c.srli, *ดูเพิ่มเติม* Shift right logical immediate, *217*
srliw, *ดูเพิ่มเติม* Shift right logical immediate word, *218*
srlw, *ดูเพิ่มเติม* Shift right logical word, *218*
sscratch, *ดู* Control and Status Register
static linking, **46**
Store
  byte, 24, *211*
  conditional
    doubleword, 124, *211*
    word, 68, *212*
  doubleword, 124, *212*
  halfword, 24, *213*
  word, 24, *219*
strip-mining, *91*
stripmining, *105*
stval, *ดู* Control and Status Register
stvec, *ดู* Control and Status Register
sub, *ดูเพิ่มเติม* Subtract, 22, 93, *ดูเพิ่มเติม* c.sub, *ดูเพิ่มเติม* Subtract, *218*
Subtract, *22*, *218*
  word, 124, *218*
subw, *ดูเพิ่มเติม* c.subw, *ดูเพิ่มเติม* Subtract word, *218*
superscalar, 3, *8*, 74
Sutherland, Ivan, 36
sw, *ดูเพิ่มเติม* c.swsp, *ดูเพิ่มเติม* c.sw, *ดูเพิ่มเติม* Store word, *219*

tail, *39*, *219*
Thoreau, Henry David, 163
Thumb-2, 60
TLB, *157*
TLB shootdown, *157*
Translation
  Mnemonic, **230**

Opcode, **230**
Translation Lookaside Buffer, *157*
Tree Sum, **222**
Turing Award
Allen, Fran, *16*
Brooks, Fred, 158
Dijkstra, Edsger W., 140
Patterson, David, ii
Perlis, Alan, *162*
Sutherland, Ivan, 36
Wirth, Niklaus, 75

User mode, *149*

vadc, *103*
vcompress, *102*
vcpop, *99*
Vector
gather, *88*
indexed load, *88*
indexed ordered load, *88*
indexed ordered store, *88*
indexed store, *88*
indexed unordered load, *88*
indexed unordered store, *88*
scatter, *88*
strided load, *87*
strided store, *87*
strip-mining, *105*
unit-stride store, *86*
unit-stride load, *86*
vectorizable, *90*, 109
vector add with carry, *103*
Vector architecture, **81**
dynamic typing, *85*, 130
type encoding, *86*
vector element index, *100*
vector length, *85*
vector length multiplier, *86*
vector length register, *85*
vector mask population count, *99*
Vector register gather, *100*
vector segment load, *105*
vector segment store, *105*
vector slide-down, *101*
vector unit-stride
fault-only-first load, *104*
vector whole-register load, *103*
vector whole-register store, *103*
vectorizable, *90*, 109
vfcvt, *97*
vfirst, 99
vfmacc, 92
vfmv, *96*
vfncvt, *97*
vfrec7, *100*
vfredosum, *99*
vfredusum, *99*
vfrsqrt7, *100*
vfslide1down, *102*
vfslide1up, *102*
vfwcvt, *97*
vid, *100*
viota, *99*
Virtual address, *154*
Virtual memory, *154*
vl, *85*, 86
vl1re32, *103*
vl2re32, *103*
vl4re32, *103*
vl8re32, *103*
vldx, *88*
vle16.v, *86*
vle16ff, *104*
vle32.v, *86*
vle32ff, *104*
vle64, 92
vle64.v, *86*
vle64ff, *104*
vle8.v, *86*
vle8ff, *104*
VLEN, **85**, 86
VLMAX, *85*, 92, 101
vlmul, *86*
vloxe16.v, *88*
vloxe32.v, *88*
vloxe64.v, *88*
vloxe8.v, *88*
vlse16.v, *87*
vlse32.v, *87*
vlse64.v, *87*
vlse8.v, *87*
vlseg2e32, *105*
vlseg3e8, *105*
vlsseg3e8, *105*
vluxe16.v, *88*
vluxe32.v, *88*
vluxe64.v, *88*
vluxe8.v, *88*
vluxseg3ei32, *105*
vma, 86
vmadc, *103*
vmand, *90*, 99
vmandn, *90*, 99
vmerge, *101*
vmnand, *90*, 99
vmnor, *90*, 99
vmor, *90*, 99
vmorn, *90*, 99
vmsbf, *100*
vmsof, *100*
vmv, *96*
vmv1r, *96*
vmv2r, *96*
vmv4r, *96*
vmv8r, *96*
vmxnor, *90*, 99
vmxor, *90*, 99
vnclip, *100*
vnclipu, *100*
vnsrl, *95*
von Neumann architecture, *13*
von Neumann, John, *13*
vredmax, *99*
vredmaxu, *99*
vredmin, *99*
vredminu, *99*
vredsum, *99*
vrgather, *100*
vse16.v, *86*
vse32.v, *86*
vse64, 93
vse64.v, *86*
vse8.v, *86*
vsetivli, *90*
vsetvl, *90*
vsetvli, *90*, 92
vsew, *86*
vslide1down, *102*
vslide1up, *102*
vslidedown, *101*
vslideup, *102*
vsoxe16.v, *88*
vsoxe32.v, *88*
vsoxe64.v, *88*
vsoxe8.v, *88*
vsse16.v, *87*
vsse32.v, *87*
vsse64.v, *87*
vsse8.v, *87*
vstx, *88*
vsuxe16.v, *88*
vsuxe32.v, *88*
vsuxe64.v, *88*
vsuxe8.v, *88*
vta, 86
vtype, *86*, 86, 91
vwmul, *95*
vwredsum, *99*
vwredsumu, *99*

Wait for Interrupt, *149*, *219*
wfi, *ดูเพิ่มเติม* Wait for Interrupt, *219*
Whole vector register move, *96*
Widening instructions, *93*, 94
William of Occam, *50*

Wirth, Niklaus, 75

x86
  aaa instruction, *4*
  aad instruction, *4*
  aam instruction, *4*
  aas instruction, *4*
  code size, 10, 132
  DAXPY, **60**
  enter instruction, 8
  Insertion Sort, **28**
  instruction reference manual
    number of pages, 14
  ISA growth, 3
  number of registers, 11
  position independent code, 11
  Tree Sum, **222**
x86-64
  AMD64, **131**
x86-64 AVX-512, 109
x86-64 AVX-512 SIMD, *107, 108*
x86-64 AVX2, 109
x86-64 AVX2 SIMD, *105*
XLEN, *141*
xnor, **121**
xor, 22, *ดูเพิ่มเติม* c.xor, *ดูเพิ่มเติม* Exclusive Or, *219*
xor properties, 23
xor register exchange, 23
xori, 22, *ดูเพิ่มเติม* Exclusive Or immediate, *219*

yield, *7*

Zba, 120, **121**
Zbb, 120, **121**
Zbc, 120, **122**
Zbs, 120, **122**
zext, **121**
Zk, **162**

คู่มือ RISC-V ฉบับภาษาไทย (Risc Five Reader, Thai language edition)

open architecture Atlas

October 22, 2018	1st edition 1st printing issued
November 12, 2018	2nd printing issued
Author	David Patterson
	Andrew Waterman
Translator	Paul Sherman & Naruemon Ratanakunakorn
Issuer	
Distributor	
Selling	

Book design & production

Printing / Bookbinding

Unauthorized copying or copying (copying, etc.) of this document is prohibited, with the exception of copyright law. Electronic data and e-books by third parties other than the purchaser are not permitted, including for private use.

For inquiries and inquiries regarding this book, please contact below.

http://riscvthai.org

© 2017 David Patterson and Andrew Waterman
Thai translation rights arranged with David Patterson and Andrew Waterman through Paul Sherman.

Printed in Thailand

ISBN 978-4-8222-9281-2